Conception d'une méthodologie d'implémentation d'applications de vision dans une plateforme hétérogène de type Smart Camera

Fabio Dias Real de Oliveira

N^o d'ordre : D.U. 2045
EDSPIC : 487

Université Blaise Pascal - Clermont-Ferrand II

École Doctorale
Sciences pour l'Ingénieur de Clermont-Ferrand

Thèse

présentée par

Fábio Dias Real de Oliveira

pour obtenir le grade de

Docteur d'Université

Spécialité : Vision pour la Robotique

Conception d'une méthodologie d'implémentation d'applications de vision dans une plateforme hétérogène de type Smart Camera

---

Soutenue publiquement le 6 juillet 2010 devant le jury :

M.	Pierre	Bonton	Président
M.	Dominique	Ginhac	Rapporteur
M.	François	Verdier	Rapporteur
M.	Stéphane	Mancini	Examinateur
M.	Pierre	Chalimbaud	Examinateur
M.	François	Marmoiton	Examinateur
M.	François	Berry	Examinateur
M.	Jocelyn	Sérot	Directeur de thèse

Résumé

Les caméras intelligentes, ou Smart Cameras, sont des systèmes embarqués de vision artificielle. Ces systèmes se différencient des caméras “communes” par leur capacité à analyser les images, afin d'en extraire des informations pertinentes sur la scène observée, et ceci de façon autonome grâce à des dispositifs embarqués de calcul. Les applications pratiques de ce type de système sont nombreuses (vidéo-surveillance, vision industrielle, véhicules autonomes, etc.), mais leur implémentation est assez complexe, et demande un haut degré d'expertise et des temps de développement élevés.

Les travaux présentés dans cette thèse s'adressent à cette problématique, et proposent une méthodologie d'implémentation permettant de simplifier le développement d'applications au sein des plateformes Smart Camera basées sur un dispositif FPGA. Cette méthodologie s'appuie d'une part sur l'instanciation au sein du FPGA d'un processeur “soft-core” taillé sur mesure, et d'autre part sur un flot de design à deux niveaux, permettant ainsi de traiter séparément les aspects matériels liés à la plateforme et les aspects algorithmiques liés à l'application.

Mots-clefs : vision par ordinateur, smart camera, FPGA, méthodologie d'implémentation, SoPC, système temps-réel, système embarqué, processeur soft-core.

Abstract

Smart Cameras are embedded artificial vision systems. These systems differ from "common" cameras due to their ability to analyze images and extract pertinent information about the observed scene, and doing this autonomously by using embedded processing resources. The application field for such systems is wide (CCTV, industrial vision, autonomous vehicles, etc.), but their implementation is quite complex and requires a high expertise level and long development times.

The work presented in this thesis deals with this problem, and proposes a design methodology which helps to simplify the application implementation process into FPGA-based smart camera platforms. This methodology is based upon a custom soft-core processor (instantiated in the FPGA), and a design flow allowing to deal separately with hardware issues dependent on the platform, and software issues dependent on the application.

Keywords: computer vision, smart camera, FPGA, design methodology, SoPC, real-time system, embedded system, soft-core processor.

Table des matières

Table des figures

Liste des tableaux

Première partie

Introduction, Motivation et État de l'art

Chapitre 1

Introduction

"La sperientia che mostra come li obbietti mandino le loro spetie ovvero similitudini intersegate dentro all'occhio nello omore albugino si dimostra quando per alchuno picholo spirachulo rotondo penetreranno le spetie delli obbietti alluminati in abitatione forte osscura: allora tu riceverai tale spetie nuna carta biancha posta dentro a tale abitatione alquanto vicina a esso spiraculo, e vedrai tutti li predetti obbietti in essa carta colle lor proprie figure e colori, ma saran minori e fieno sottosopra per chausa della detta interseghatione; li quali simulacri se v'ussciranno del loco alluminato del sole saran proprio dipenti in essa carte, la qual vol essere sottilissima e veduta da rivescio, e lo spirachulo detto sie fatto in piastra sottilissima di ferro."

"Voici l'expérience qui nous apprend la manière dont les objets envoient leurs images se croiser sur l'humeur albugineuse au dedans de l'oeil. Lorsque les images des objets éclairés pénètrent par un petit trou rond dans un appartement très obscur, recevez ces images dans l'intérieur de l'appartement sur un papier blanc situé à quelque distance du trou, vous verrez sur le papier tous les objets avec leurs propres formes et couleurs ; ils seront diminués de grandeur ; ils se présenteront dans une situation renversée, et cela en vertu de l'intersection déjà indiquée. Si les images viennent d'un endroit éclairé par le soleil, elle vous paraîtront comme peintes sur le papier qui doit être très mince, et vu par derrière. Le trou sera pratiqué dans une plaque de fer aussi très mince."

Leonardo da Vinci, artiste, ingénieur et scientifique italien du XVe siècle.
Manuscrit D, folio 8.
Description et comparaison de la formation des images inversées dans l'oeil et dans la "camera obscura", aïeule des dispositifs d'acquisition d'images.
Extrait de [1], traduction française extraite de [2].

La vision.

Parmi toutes les capacités perceptives de l'être humain, et des animaux en général, la vision est sans doute la plus puissante. Grâce à la vision nous sommes capables de nous guider dans un environnement inconnu, reconnaître des objets et des personnes, éviter des obstacles en mouvement, et réaliser d'innombrables autres tâches. D'un point de vue fonctionnel ces tâches peuvent s'avérer très complexes, même si nous les exécutons quotidiennement de façon parfaitement automatique.

Le système visuel humain est capable de s'adapter à différentes conditions de luminosité, et de fonctionner dans une gamme de distances allant de quelques centimètres à quelques kilomètres, tout en fournissant des information telles que couleur, forme, relief, distance, vitesse, etc. Toutes ces caractéristiques font de la vision le capteur idéal (ou plutôt le système perceptif idéal) pour toutes les tâches concernant la mobilité, la localisation et la reconnaissance/identification.

Mais comment cela est-il possible ? Depuis plusieurs siècles l'homme s'interroge sur les propriétés étonnantes de son propre système visuel, ainsi que sur les relations entre l'oeil, l'image et la lumière. Déjà au Ier siècle le philosophe Sénèque constate et décrit, en observant un objet à travers un ballon de verre rempli d'eau, que l'image de ceci était grossie. Depuis, bon nombre de découvertes et inventions ont vu le jour, aussi bien dans le domaine de la physiologie de la vision que dans le domaine de l'optique.

Ces découvertes et inventions ont permis de mieux comprendre le processus visuel biologique, et ensuite de l'améliorer et le reproduire. Mais au-delà des domaines de la vision et de l'optique en eux-mêmes, ces avancées ont permis surtout d'accroître radicalement notre connaissance et compréhension du monde et de l'univers. Nous sommes donc devenus capables de voir plus loin grâce aux lunettes astronomiques (Hans Lippershey en 1608, puis Galilée en 1609), et de voir "l'invisible" grâce aux microscopes (inventé au XVIIe siècle)(figure 1.1).

Nous sommes aussi devenus capables de "capturer" des images. La *"Camera obscura"* aïeule de l'appareil photo, a été clairement décrite par Leonard da Vinci au début du XVIe siècle [1] [2], mais son invention date sûrement de bien avant. Le phénomène de l'obtention d'une image inversée après le passage de la lumière par un petit orifice (sténopé ou *pinhole*) était connu depuis bien longtemps, et a été décrit par le philosophe chinois Mo-Ti au Ve siècle av. J.-C., et par le philosophe grecque Aristote au IVe siècle av. J.-C.

Et une fois capables de capturer des images, nous apprîmes ensuite à les reconstituer, aboutissant à l'art de la photographie, et plus tard, du cinéma.

Et au XXe siècle, une nouvelle barrière a été franchie. La vision a quitté le domaine de l'optique et de la physiologie, pour passer du coté d'un autre domaine en plein essor : la micro-électronique.


FIG. 1.1 – *Observations de la lune par Galilée en 1616, et microscope du XVIIIe siècle.* Source des images : Wikipedia.

1.1 Le traitement d'images et la vision artificielle : contexte

En grande partie grâce aux travaux d'Albert Einstein sur l'effet photoélectrique, qui lui ont valu le prix Nobel de physique en 1921, l'image au XXe siècle ne s'exprime plus exclusivement en lumière. L'image peut aussi s'exprimer en électrons, bits et finalement, pixels. Et c'est grâce à cette nouvelle forme de représentation de l'image que des nouvelles branches de la science sont nées, parmi elles le traitement d'images et la vision artificielle ou vision par ordinateur.

Dans les dernières décennies, avec l'évolution des processus technologiques permettant la démocratisation et la miniaturisation des systèmes d'acquisition et traitement de l'information, la vision artificielle et l'imagerie numérique ont gagné de plus en plus d'importance dans les cercles scientifique et industriel, ainsi que dans la vie de tous les jours. A tel point qu'aujourd'hui en France, et plus généralement dans tous les pays développés, la majorité de la population est quotidiennement équipé d'un système numérique d'acquisition et traitement d'images, le plus souvent intégré dans leur téléphone portable.

La dissémination des ordinateurs personnels (ou PCs, de *Personal Computer*), et des périphériques associés à l'image tels que les scanners et webcams, a amplifié l'intérêt du grand public. Il en va de même pour l'explosion du phénomène Internet, avec ses nouvelles formes de communication incluant l'envoi et le partage facile et rapide d'images, et la communication temps réel par vidéo. Aussi, les logiciels de traitement du type PhotoShop se sont largement popularisés, rassemblant des utilisateurs issus de tous les publics, et non exclusivement des traiteurs de signaux ou informaticiens chevronnés.

Nous pouvons aussi citer la course industrielle vers l'automatisation, qui a provoqué le rapide développement du domaine de la robotique, fort demandeur en systèmes de perception efficaces. Or, comme dit précédemment, pour les tâches concernant la localisation et la mobilité, la perception visuelle peut être de grande aide, pouvant s'adapter à une large palette d'applications et fournissant des données de différents types, aussi bien sur le système sur lequel le dispositif de vision est installé (données proprioceptives), que sur l'environnement dans lequel ce système évolue (données extéroceptives).

En outre, la généralisation des examens médicaux par imagerie (notamment l'IRMN, de Imagerie par Résonance Magnétique Nucléaire), et plus récemment les interventions chirurgicales à l'aide de la réalité augmentée, créent des nouveaux domaines d'application pouvant bénéficier de façon extraordinaire des progrès techniques de la vision par ordinateur.

Nous nous retrouvons donc dans un scénario où il existe une très forte demande pour les systèmes de vision artificielle et de traitement d'images, aussi bien du coté industriel (automatisation et contrôle de qualité) comme de la part de la société civile (vidéo-surveillance, reconnaissance de visages, cinéma, photographie numérique, imagerie médicale) et des structures militaires (Unmanned Aerial Vehicles (UAV's), véhicules autonomes, télé-guidage, etc.). Souvent, ces systèmes doivent respecter des fortes contraintes de taille, autonomie, consommation d'énergie et performances.

D'un autre coté, les avancés en micro-électronique fournissent chaque année des nouveaux outils et dispositifs rendant possible la conception de systèmes de vision artificielle de plus en plus performants, et capables de respecter les contraintes imposées. Tous les éléments sont donc réunis pour faire de la vision artificielle un des "challenges" scientifiques les plus prometteurs, voire fédérateurs, de notre époque. Ceci parce que le développement d'un système de vision demande des connaissances appartenant à plusieurs disciplines, du traitement du signal à l'architecture des ordinateurs, en passant par les théories des probabilités, l'algèbre linéaire, l'informatique, l'intelligence artificielle, et l'électronique analogique et numérique. Cette pluridisciplinarité rend la tâche d'autant plus compliquée, et les retombées scientifiques d'autant plus importantes.

1.2 Du paradigme de Marr à la vision active.

Au début, entre les années 20 et 70, le traitement d'images était essentiellement destiné à préparer les images pour leur transmission, ou pour leur analyse visuelle par un être humain. Ces traitements pouvaient se classifier en trois catégories :

- Restauration : correction des imperfections et défauts apportés par le processus d'acquisition. Exemples : filtrage du bruit, correction du contraste.

- Amélioration : rendre l'image plus belle, ou plus adaptée à une analyse à l'oeil nu afin d'étudier un phénomène. Exemples : expansion de dynamique, rehaussement des contours.

- Compression : réduire le volume en données de l'image afin de permettre ou faciliter son stockage et son envoi par des moyens de télécommunication.

Quelques applications étaient l'analyse d'images aux rayons X, les images radar, les systèmes de communication par fax, l'analyse d'images issues des chambres à bulles[1], etc. Le point commun est le fait que le traitement d'images ici a pour seul but de corriger, améliorer ou "transporter" l'image, toujours en ayant en vue son exploitation postérieure par un observateur humain. Il ne s'agit pas pour le moment d'extraire automatiquement des informations sur la scène observée. Nous pouvons donc parler de traitement d'images, mais pas encore de vision artificielle.

Dans les années 70 une évolution naturelle vers l'extraction automatique d'informations a eu lieu, avec l'apparition des notions de description structurelle de la scène. Des nouveaux types de traitements ont émergé, comme le seuillage, l'extraction de contours et la morphologie mathématique. Parallèlement, le développement de "l'intelligence artificielle" [4], sorti des pages de la science fiction vers les pages des journaux scientifiques, et suivi de la création de systèmes experts pour différentes applications, a donné une impulsion au développement d'applications d'identification et interprétation par vision.

A la fin des années 70, le chercheur anglais en neuroscience David Marr, travaillant au MIT, a défini un paradigme posant des bases conceptuelles pour la vision artificielle [5]. Selon le paradigme de Marr, l'objectif d'un système de vision est de reconstruire la scène 3D observée par la caméra. Cela veut dire récupérer la dimension qui est perdue lors de la projection de la scène tridimensionnelle sur le plan image 2D. Cette reconstruction est faite au moyen d'une chaîne linéaire de traite-

1. Chambre à bulles : détecteur de particules très utilisé dans les années 50 et 60, inventé par Donald Glaser en 1952 [3], et lui valant le prix Nobel de la physique en 1960.

ments de différents niveaux, décomposée en trois étapes comme décrit ci-dessous et illustré en figure 1.2 :

- Première étape : basée sur des traitements bas-niveau appliqués pixel par pixel ou mettant en relation un pixel et son voisinage immédiat dans l'image. L'objectif principal de ces traitements est de segmenter l'image dans des sous-ensembles de pixels représentatifs des caractéristiques de la scène et des objets observés (coins, bords, ombres, contours, etc.). Le résultat obtenu après l'application des traitements de bas-niveau est un ensemble de primitives 2D, constituant une première ébauche de la scène.

- Deuxième étape : à partir des primitives 2D obtenues par les traitements de bas-niveau, des traitements de moyen-niveau sont appliqués afin d'extraire une première représentation spatiale de la scène, sans pour autant représenter une reconstruction 3D. Le résultat obtenu est communément appelé une ébauche 2,5D, car même si des informations relatives à la profondeur de la scène sont retrouvées, ces informations restent insuffisantes pour décrire la localisation des objets dans l'espace tridimensionnel. Il s'agit plutôt d'une carte de profondeurs relatives entre les différentes primitives 2D observés et l'observateur lui-même (caméra). Ces traitements sont essentiellement basés sur la mise en correspondance des différentes primitives, et peuvent utiliser des transformations géométriques basées sur des modèles de projection. Cette étape est fort dépendante du modèle et du calibrage de la caméra utilisée (ou des caméras dans le cas d'un système stéréoscopique).

- Troisième étape : des traitements de moyen et haut-niveau viennent intégrer les connaissances à priori de la caméra et de la scène 3D à la carte de profondeurs obtenue précédemment, résultant finalement dans un modèle tridimensionnel reconstruit centré sur la scène, et indépendant donc du point de vue (position de l'observateur).

Ce cadre théorique a provoqué un fort engouement lors de son apparition, et plusieurs travaux scientifiques visant la reconstruction d'une scène 3D à partir d'un ensemble d'images à vu le jour. Plusieurs méthodes ont été proposées, les plus répandues étant les méthodes par stéréoscopie, les méthodes *Structure from Motion* (SfM), et les méthodes *Shape from Shading* (SfS).

- La stéréoscopie exploite des images prises par différentes caméras, afin de reconstituer le relief de la scène en utilisant la géométrie épipolaire. Cette approche est inspirée de l'appareil visuel humain, capable d'apercevoir la profondeur d'une scène à partir de la combinaison des images perçues par les deux yeux.

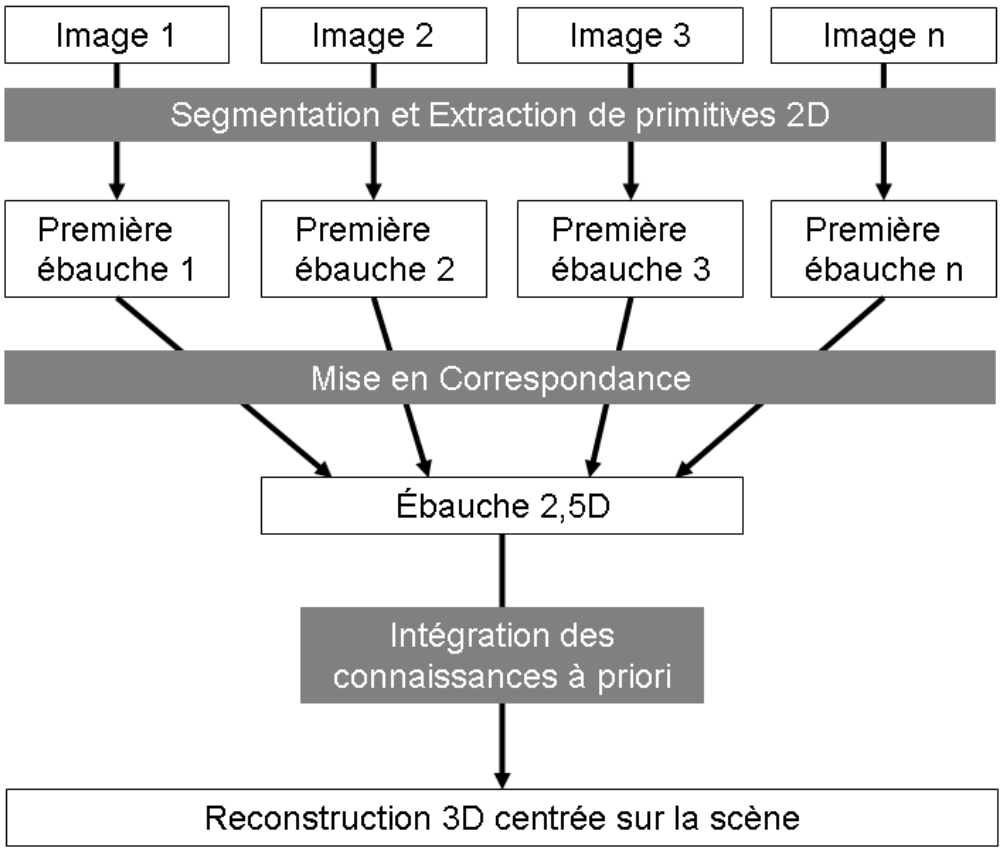


FIG. 1.2 – *Description des étapes de traitement proposées dans le paradigme de Marr.*

– Les méthodes *Structure from Motion* utilisent plusieurs images prises par une même caméra, et exploitent les mouvements de la caméra ou de la scène afin d'obtenir les différentes prises de vue nécessaires à la reconstruction [6] [7].

– Les méthodes *Shape from Shading* utilisent une seule image pour reconstituer le relief de la scène, et sont soumises à des fortes contraintes liées à l'éclairage et aux caractéristiques des surfaces [8] [9].

Malgré l'apparition de plusieurs systèmes et méthodes basés sur le paradigme de Marr, et même si ce paradigme est encore d'actualité pour quelques applications, le bilan de ce cadre théorique est plutôt mitigé, voire décevant. Peu de problèmes ont été résolus, peu d'applications concrètes ont été retrouvées, et cette formalisation théorique de la vision a été peu à peu remplacée par d'autres paradigmes.

En effet, le paradigme de Marr s'adresse de façon générale à la reconstruction et à la description spatiale et tri-dimensionnelle d'une scène observée. Or, voir n'est pas uniquement être capable de décrire une scène. La reconnaissance, l'identifica-

tion et beaucoup d'autres tâches visuelles font appel à des processus cognitifs et psychologiques qui ne sont pas uniquement liés aux propriétés géométriques d'un objet.

Une autre explication pour le remplacement graduel du paradigme de Marr par d'autres approches et paradigmes est liée à la capacité de traitement des machines. En effet, il y a 20 ans les ordinateurs n'avaient pas la puissance de calcul des machines d'aujourd'hui. L'approche de Marr étant basée sur des traitements appliqués sur l'ensemble des pixels d'une image, ceci engendre un grand nombre de calculs à exécuter sur des quantités de données assez importantes. La mise en relation entre chaque région de l'image et son voisinage proche pour extraire des primitives, suivie de la mise en relation des différentes primitives afin d'en extraire des objets, provoque rapidement une explosion combinatoire, qui rendait l'exécution de telles méthodes extrêmement difficile et lente pour les machines des années 80. D'ailleurs, même avec la puissance de calcul offerte par les machines modernes, ces tâches restent encore loin d'être aisées, rendant les implémentations en temps-réel très difficiles.

Afin d'offrir une alternative au paradigme du traitement des images par une chaîne linéaire de processus visant la reconstruction, d'autres approches et paradigmes ont été proposés vers le milieu et fin des années 80. Nous pouvons citer les systèmes multi-agents, brièvement présentés ci-dessous, et la vision active, présentée en détails dans la prochaine section.

La proposition principale des systèmes multi-agents consiste dans la coopération par l'échange d'informations entre les différents modules, ou agents, du système. Différemment du paradigme de Marr, où les différents modules de traitement sont indépendants, sans retour d'information des niveaux plus élevés vers le bas-niveau, le système multi-agents propose une approche guidée par la tâche à exécuter, prenant en compte les caractéristiques intrinsèques des agents (traitements) mis en jeu, ainsi que le type d'image à analyser. Un système superviseur peut être utilisé, afin de sélectionner et coordonner le fonctionnement des agents, selon le sous-problème à résoudre et basé sur des connaissances et règles opératoires attachées à la résolution de ce problème. Par contre, le recours à un superviseur n'est pas obligatoire, celui-ci pouvant être remplacé par une stratégie de contrôle distribué. Nous verrons que cette approche guidée par la tâche (task-driven) et le retour d'information (feedback) entre les différents niveaux font également partie des principaux postulats de la vision active.

Nous pouvons citer comme exemples de systèmes multi-agents le système KISS [10] (Knowledge-based Image Segmentation System) et l'environnement ADAGAR [11] (Atelier de Développement d'AGents sur Architecture Répartie). Le premier est basé sur des agents de deux types : les KS (Knowledge Server) et les KP (Knowledge Processor). Ces agents sont interconnectés suivant l'architecture présentée en figure 1.3, et communiquent par envoi de messages. L'environnement ADAGAR exploite une implémentation en architecture distribuée (multi-processeurs), et est basé sur le

modèle *blackboard*, ou tableau noir (hiérarchisation des agents, contrôle distribué). Il s'agit essentiellement d'un environnement de développement d'applications modulaires et extensibles, permettant la génération de nouveaux agents et la gestion de leur exécution.

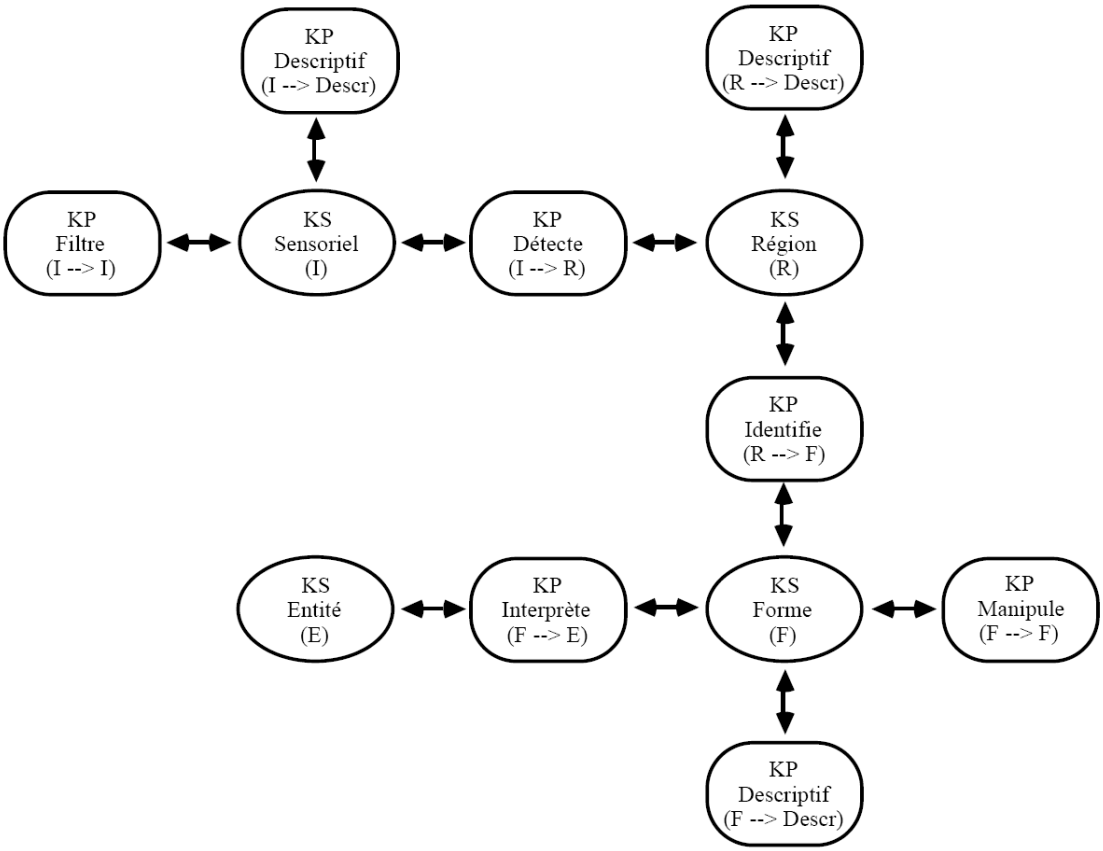


FIG. 1.3 – *Vue d'ensemble du système KISS.*

1.3 Vision active

Comme vu précédemment, au début des années 80 les efforts de recherche dans le domaine de la vision artificielle ont été concentrés sur la transformation des données bidimensionnelles des images en représentation tridimensionnelle d'une scène. L'objectif était d'inférer les surfaces, volumes, contours, ombres, occlusions, distances et mouvements. Mais plusieurs tentatives visant à obtenir une représentation complète d'une scène 3D ont échoué, et encore aujourd'hui cette tâche reste bien difficile, même avec nos moyens informatiques de calcul, bien plus évolués que ceux de l'époque de David Marr.

La vision active est apparue comme une approche alternative pour traiter les

problèmes de la vision artificielle. L'idée centrale de la vision active est de prendre en compte l'aspect perceptif des tâches visuelles, en s'inspirant des systèmes de vision biologiques, notamment le système humain. Au lieu d'obtenir une représentation spatiale de la scène, l'objectif d'un système de vision active est d'extraire uniquement l'information nécessaire pour accomplir une certaine tâche, ou résoudre un problème donné. Ceci donne lieu à une stratégie d'observation guidée par la tâche (*task-driven*).

Plusieurs groupes de chercheurs se sont penchés sur la problématique, le développement et les applications de la vision active. Les travaux les plus influents sont *Active Vision* de Yiannis Aloimonos en 1987 [12], *Active Perception* de Ruzena Bajcsy en 1988 [13] [14] , *Animate Vision* de Dana Ballard en 1989 [15] [16], et *Purposive Qualitative Vision* de Yiannis Aloimonos en 1990 [17]. Ces travaux sont brièvement résumés ci-dessous :

Active Vision, par Yiannis Aloimonos, 1987

Ce travail [12] est couramment considéré comme le pionnier [2] dans le cadre de la vision active. Son idée centrale consiste à prendre en compte le mouvement du capteur d'images afin de lever certaines ambiguïtés, permettant ainsi de résoudre plus facilement un certain nombre de problèmes de vision.

Selon Aloimonos, *"un observateur est considéré actif quand il est engagé dans un type quelconque d'activité dont le but est de contrôler les paramètres géométriques de l'appareil sensoriel. L'objectif de cette activité est de manipuler les contraintes adjacentes au phénomène observé, afin d'améliorer la qualité des résultats perceptifs"*.

Néanmoins, ce travail reste concentré sur les aspects mathématiques (linéarité, stabilité et unicité des solutions) des méthodes telles que le *Structure from Motion* et le *Shape from Shading*. Ci-dessous (tab. 1.1) est reproduit un tableau présenté à [12], décrivant les avantages liés à un observateur actif dans la résolution de quelques problèmes classiques de vision par ordinateur :

Active Perception, par Ruzena Bajcsy, 1988

L'*active perception* de Ruzena Bajcsy [13] [14] est la première approche à intégrer pleinement le capteur d'images et le contrôle de ses paramètres dans la boucle de

2. Aloimonos explicite néanmoins dans son papier que le travail de recherche présenté a bénéficié de discussions avec Dana Ballard et Ruzena Bajcsy, qui présenteraient eux-mêmes d'autres approches de la vision active. Il est à remarquer que Ruzena Bajcsy avait déjà présenté l'*Active Perception* en 1985 [18].

Problème	Observateur Passif	Observateur Actif
Shape from Shading	Problème mal posé, a besoin d'être régularisé. Même régularisé, l'obtention d'une solution unique n'est pas garantie à cause de la non-linéarité.	Problème bien posé. Solution unique. Utilisation d'équations linéaires. Stabilité.
Shape from Contour	Problème mal posé. N'a pas encore été régularisé dans le sens de Tichonov. Résoluble sous des hypothèses restrictives.	Problème bien posé. Solution unique aussi bien pour un observateur monoculaire que binoculaire.
Shape from Texture	Problème mal posé. Requiert des hypothèses sur la texture.	Problème bien posé. Pas d'hypothèse nécessaire.
Structure from Motion	Problème bien posé mais instable. Contraintes non-linéaires.	Problème bien posé et stable. Contraintes quadratiques, méthodes de solution simples, stabilité.
Flot Optique	Problème mal posé, a besoin d'être régularisé. L'introduction du lissage peut produire des résultats erronés.	Problème bien posé. Solution unique. Peut être instable.

TAB. 1.1 – *Tableau proposé par Yiannis Aloimonos démontrant l'apport d'un observateur actif dans la résolution de différents problèmes de vision.*

perception. Cette approche est définie comme étant une étude des stratégies de contrôle et modélisation de la perception.

La stratégie consiste à définir un ensemble d'actions et paramètres du système sensitif, afin d'extraire de la scène l'information la plus pertinente pour résoudre un certain problème. La définition de ces actions et paramètres dépend d'une part d'un ensemble de modèles locaux, décrivant le comportement du capteur et des modules de traitement associés, et d'autre part d'un modèle global définissant l'interaction entre le capteur et ces différents modules.

L'ensemble des paramètres et actions peut donc être retrouvé par la minimisation d'une fonction de coût, mettant en relation l'information obtenue en fonction des différents paramètres et états du système d'acquisition. Ce cadre théorique a donné lieu à des expérimentations au moyen de la plateforme "Agile", illustrée en figure 1.4. Cette plateforme est un système binoculaire, avec 11 degrés de liberté. Une des applications implémentées a été l'obtention de cartes de profondeur.

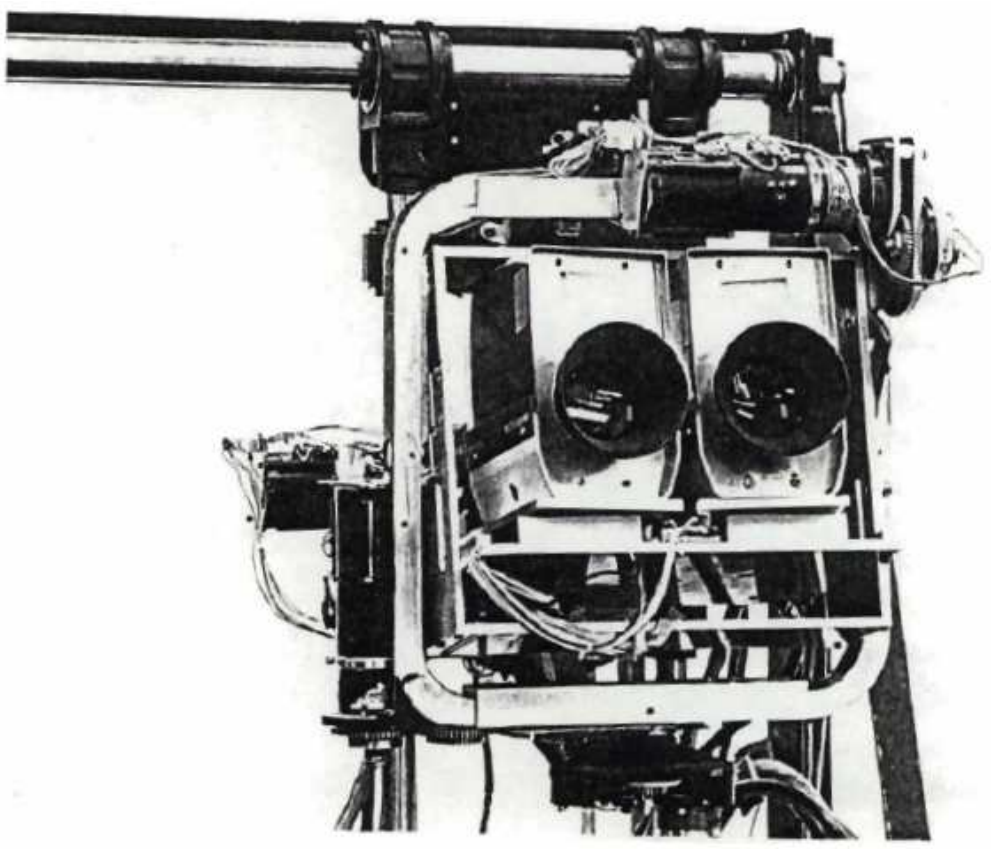


FIG. 1.4 – *Le système binoculaire "Agile Camera System". University of Pennsylvania, 1988.*

Pendant que l'*Active Vision* de Yiannis Aloimonos s'intéresse surtout aux conséquences, d'un point de vue mathématique, d'un observateur actif dont le mouvement est connu, l'*Active Perception* du Prof. Bajcsy se concentre justement sur l'activité de l'observateur (capteur), afin d'optimiser le processus d'acquisition de données.

Animate Vision, par Dana Ballard, 1989

La vision animée de Dana Ballard [15] [16] propose la conception d'un système perceptif bio-inspiré, notamment en ce qui concerne la résolution hétérogène de la rétine (on parle de système fovéal), et le mécanisme de vision par saccades, qui sera expliqué plus en détails dans le chapitre 2. L'idée centrale de cette approche est de contrôler le mouvement du regard de façon "intelligente", en s'adaptant dynamiquement à l'évolution du système et de l'environnement.

En effet, l'*Animate Vision* vient souligner l'aspect cognitif de la vision, introduisant des concepts tels que l'apprentissage, et prenant en compte la morphologie de

l'observateur, ainsi que celle de l'environnement. Un de ses principaux concepts est l'utilisation d'un système de coordonnées exocentriques, i.e. centré sur l'objet et non sur l'observateur. L'avantage d'un tel repère centré sur l'objet est son invariance par rapport aux mouvements de l'observateur.

Les travaux de Ballard commencent à suggérer la division des tâches visuelles en étapes d'attention, focalisation et identification (haut-niveau), notamment grâce à la notion de "gaze control" (contrôle du regard). **Cette structuration du processus visuel, comme il sera expliqué dans la suite, est un des principaux concepts exploités dans le cadre de cette thèse**. Néanmoins, les travaux de Ballard sont spécialement concentrés sur le cas d'une plateforme de vision anthropomorphique (tête robotique binoculaire à résolution fovéale), n'étant pas forcément approprié à d'autres types de système.

Purposive Qualitative Vision, par Yiannis Aloimonos, 1990

La vision intentionnelle et qualitative [17] est un deuxième paradigme proposé par Yiannis Aloimonos, qui vient d'une certaine façon compléter celui de la *Vision Active* proposé quelques années auparavant. Le concept principal de cette approche vient de l'observation que, dans la grande majorité des cas, un système de vision donné a pour but de résoudre un problème bien spécifique. La spécification détaillée et précise du problème à résoudre permet de guider le processus d'acquisition de données, ainsi que les traitements à effectuer sur ces données afin de récupérer l'information désirée et d'accomplir une certaine tâche.

L'application donnée comme exemple dans [17] est la navigation autonome d'un système doté de capteurs d'images. L'objectif est de démontrer qu'un tel système est parfaitement capable d'accomplir des tâches de navigation dans l'environnement, sans pour autant procéder à une étape de reconstruction. La plateforme expérimentale utilisée est le système MEDUSA, composé d'un système actif d'acquisition d'images, de capteurs inertiels et d'un bras manipulateur visible dans le champ de la caméra. Cet ensemble est capable de se déplacer dans l'environnement.

Le "cerveau" du système (figure 1.5) est composé par un ensemble d'unités de traitement, chacune responsable de l'exécution d'une tâche bien précise :

- calcul du flot optique à partir d'une séquence d'images (**2-D** dans la figure) ;
- détection et localisation d'objets mouvants dans la scène (objets dont le mouvement est indépendant de celui de MEDUSA) (**A**) ;
- détection des parties de la scène s'approchant de MEDUSA (soit par mouvement propre ou par conséquence des mouvements du système) (**B**) ;
- suivi des objets mouvants dans le champ de vision (**C**) ;
- détection si un objet mouvant est en route de collision avec MEDUSA? (**D**) ;
- contrôle des moteurs afin d'intercepter un objet (**E**) ;

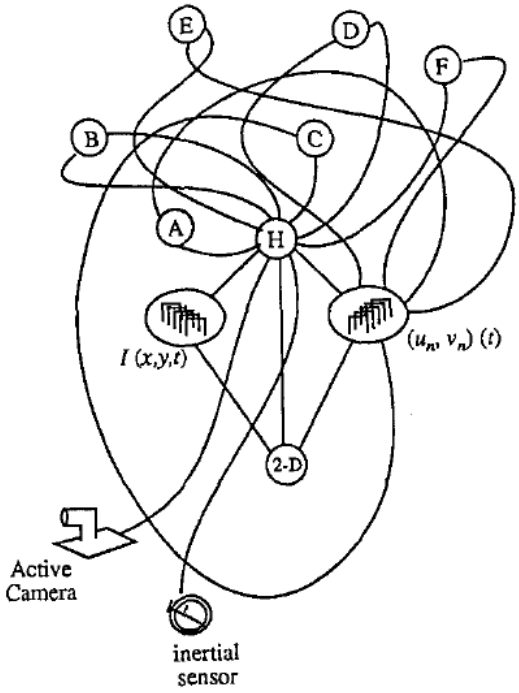


FIG. 1.5 – *Description schématique du système MEDUSA.*

Toutes les unités de traitements sont reliées à une unité de contrôle central (**H** dans la figure 1.5), dont le but est de synchroniser les tâches et relayer les résultats obtenus. Cette unité de contrôle détient la connaissance de la tâche à exécuter, et est capable de déterminer quels sont les processus à exécuter, et dans quel ordre, pour arriver au résultat souhaité. Dans la figure 1.5, $I(x,y,t)$ représente les images acquises (entrée de données du système), et $(Un,Vn)(t)$ représente le flot optique normal calculé par le module **2-D**.

Le fait de connaître précisément le problème permet de le décomposer en sous-tâches plus simples, qualitatives, sans avoir recours à une reconstruction complète de la scène qui contiendrait un excès d'information par rapport à celle effectivement nécessaire. C'est l'essence même du paradigme de la vision intentionnelle, agir en fonction du problème. L'inconvénient de cette approche par contre réside exactement dans cette spécialisation du système en fonction du contexte : un léger changement du cadre d'application peut en effet demander une restructuration totale du système. Il est à remarquer également que ce paradigme s'applique aussi bien aux systèmes monoculaires que binoculaires.

Les quatre travaux présentés dans les paragraphes précédents composent les bases de ce que nous appelons communément aujourd'hui la *Vision Active*. Ces quatre approches sont à la fois redondantes et complémentaires, chacune mettant l'accent sur une caractéristique différente d'un même concept général. Le plus souvent, ces cadres théoriques ont trouvé des applications dans le domaine de la robotique, notamment dans l'utilisation de systèmes binoculaires embarqués dans une tête robotique [19] ou dans un robot mobile [20].

Bien d'autres chercheurs, issus de différents domaines scientifiques, se sont intéressés au cadre proposé par la vision active. Nous pouvons ainsi citer les travaux liés aux neurosciences et à la psychologie, qui discutent des mécanismes cognitifs associés aux processus visuels biologiques (humains ou animaux) [21], et proposent des solutions pour leur implémentation au sein d'une machine [22]. D'autres travaux, issus du domaine de l'informatique, proposent des systèmes opérationnels [23], ou des outils de programmation [24] pour les applications de vision active.

Évidemment, les travaux issus de la communauté vision par ordinateur sont aussi très nombreux, aussi bien d'un point de vue théorique [25], que du point de vue applicatif. Nous citerons notamment les travaux de John Konstantine Tsotsos et al. [26] [27], de François Chaumette et Eric Marchand [28], de Thierry Viéville [29] et le livre de Andrew Blake et Alan Yuille intitulé *Active Vision* et publié en 1993 [30].

Les aspects matériels et architecturaux des systèmes de vision active sont le sujet de la thèse soutenue par Pierre Chalimbaud en 2004 [31]. La thèse ici présentée peut être considérée comme la suite de ces travaux.

1.3.1 Conclusions sur la vision active et objectif de cette thèse

D'après les différents travaux cités tout au long de cette section, il est possible de constater qu'une des caractéristiques essentielles de la vision active est la rétroaction. Celle-ci consiste dans une boucle de retour d'informations permettant l'adaptation dynamique du processus d'acquisition de données. Cette adaptation est pilotée en fonction de l'état actuel du système et de la tâche à exécuter, renforçant l'aspect intentionnel de la vision. Dans les systèmes de vision artificielle, cette rétroaction peut s'exprimer à différents niveaux :

- Mécanique : mouvements de la caméra (pan, tilt), ou du système portant la caméra.

- Optique : zoom, focus, contrôle de l'ouverture du diaphragme.

– Électrique / Électronique : contrôle du processus d'acquisition, fenêtrage, contrôle de la conversion analogique/numérique, adaptation du temps d'intégration.

– Algorithmique : stratégie d'observation, traitement adaptatif, paramétrage des traitements appliqués.

Une des originalités du travail présenté par Pierre Chalimbaud [31] et repris ici est d'exploiter de façon approfondie les aspects électriques et électroniques de la rétroaction sur des systèmes monoculaires. L'exploitation de ces aspects requiert un contrôle très fin du dispositif d'acquisition d'images. Nous soutenons qu'une telle finesse dans le contrôle du capteur ne peut être obtenue qu'à l'aide d'un système de traitement embarqué au plus proche de l'imageur, au sein même de la caméra. Ceci permet d'obtenir un lien "privilégié" avec le capteur d'images, impossible à effectuer via les bus industriels standards et protocoles associés (USB, Firewire, Ethernet). Ces derniers introduiraient des latences inadmissibles pour le contrôle dynamique du capteur, sans même parler du non déterminisme de certains protocoles qui rendraient la synchronisation des tâches impossible ou inefficace.

En outre, la vision active présente des caractéristiques fort contraignantes pour l'architecture du système. Par ailleurs, l'aspect intrinsèquement parallèle de certaines tâches rend les architectures classiques de type PC inadaptées. Une fois de plus, l'utilisation d'une architecture dédiée se justifie.

Cette architecture dédiée se compose idéalement d'une plateforme unique, comportant aussi bien l'appareil sensoriel que des structures de calcul. L'objectif est alors de réaliser les tâches propres de la vision précoce (attention et focalisation) au sein même de la caméra et avant transmission vers le système hôte, qui sera lui en charge des traitements de haut-niveau. Ce type de structure est communément appelée *Caméra Intelligente*, ou *Smart Camera* selon le terme consacré en anglais.

Si les travaux de P. Chalimbaud ont conduit à proposer, puis à réaliser une telle plateforme du type Smart Camera, ils laissaient ouverts les aspects liés à sa programmation, laquelle était réalisée à un niveau d'abstraction assez bas. L'objectif de cette thèse est dés lors de proposer une méthodologie de développement, ainsi que les outils de programmation associés, pour les applications de vision précoce implantées sur les architectures embarquées et reconfigurables, ceci afin de pallier la difficulté d'implémentation de ces applications au sein de ce type de plateforme.

1.4 Organisation du manuscrit

Ce manuscrit est organisé de la façon suivante :

Chapitre 2 - Motivations et Problématique

Ce chapitre traitera des motivations du travail réalisé dans cette thèse sous un point de vue théorique. A partir du concept de vision active, présenté en introduction, le concept de vision précoce sera discuté, et la structuration des processus visuels en tâches d'attention, focalisation et interprétation sera justifiée.

Ensuite, nous aborderons les thèmes de l'attention visuelle, des cartes de saillance, de la vision par saccades et des routines visuelles. A partir de cette base théorique, nous aborderons finalement les aspects architecturaux et matériels des systèmes de traitement d'images, en analysant quelles sont les contraintes posées par la vision précoce. Ceci nous amènera naturellement aux aspects de la vision embarquée et au concept de caméra intelligente (*Smart Camera*).

Chapitre 3 - État de l'art : Smart Cameras

Dans ce chapitre il sera fait un état de l'art sur les systèmes embarqués de traitement d'images temps-réel. Tout d'abord les différents dispositifs pouvant constituer de tels systèmes seront brièvement présentés, à savoir les dispositifs sensoriels (capteur CCD, capteur CMOS, capteurs inertiels), les dispositifs de communication (interface USB, Firewire, Ethernet, Camlink), et les dispositifs de traitement de l'information. Ces derniers sont classifiés en dispositifs de type fixe (ASIC), programmables (processeurs embarqués de type ARM, DSP, processeurs média) et reconfigurables (CPLD / FPGA et processeurs soft-core).

Ensuite, différents types de "Smart Cameras" seront présentés et analysés. Nous nous concentrerons notamment sur les aspects matériels et architecturaux. Seront donnés en exemple des dispositifs provenant aussi bien du milieu industriel qu'académique.

Finalement, la plateforme SeeMOS, développée initialement par P. Chalimbaud et qui a été utilisée comme support expérimental pour cette thèse, sera présentée en détail. Les caractéristiques d'un tel système viendront susciter le besoin d'une méthodologie de conception dédiée, afin d'implémenter efficacement des applications de vision dans ce type de plateforme (hétérogène / reconfigurable).

Chapitre 4 - État de l'art : Méthodologies de développement

Dans ce chapitre il sera dressé un état de l'art sur les méthodologies de développement d'applications pour les systèmes embarqués, et notamment pour les systèmes basés sur composants reconfigurables du type FPGA. L'objectif est de s'appuyer sur les solutions existantes afin d'en extraire les éléments permettant de proposer une approche originale, dédiée aux architectures du type caméra intelligente basées sur FPGA.

Chapitre 5 - Méthodologie proposée

La méthodologie proposée pour l'implémentation d'applications de vision précoce sur des plateformes du type caméra intelligente sera présentée ici. Cette méthodologie repose essentiellement sur la définition d'un processeur sur mesure capable d'harmoniser le contrôle de tous les composants de la plateforme sous un seul jeu d'instructions. Ainsi, un seul et unique code assembleur contiendra toutes les informations nécessaires pour commander et synchroniser capteurs, mémoires, interfaces et unités de calcul. L'architecture d'un tel processeur sera décrite, et la définition du jeu d'instructions expliquée.

On démontrera que cette homogénéisation par software d'un hardware fortement hétérogène facilite grandement la tâche de programmation et d'implémentation.

La mise en oeuvre de cette méthodologie s'appuie sur un modèle descriptif virtuel en langage SLDL (System Level Design Language), d'une part du coeur du processeur (décodeur d'instructions, *register file*, ALU), et d'autre part des composants périphériques constituant la plateforme (blocs mémoire, capteurs, unités de calcul spécialisées). Cette modélisation permet la simulation, le paramétrage et le débogage de l'ensemble du système (processeur + composants hardware périphériques + code application), sans pour autant procéder à la synthèse du design à chaque itération. La synthèse n'est effectuée qu'une fois l'application validé d'un point de vue logiciel (débogage du code assembleur), et d'un point de vue matériel (simulation du fonctionnement du système).

Chapitre 6 - Implémentation dans la plateforme SeeMOS

Ce chapitre décrit en détail l'application de la méthodologie proposée à la plateforme SeeMOS. Le contrôle des différents dispositifs matériels à partir du jeu d'instructions présenté précédemment sera expliqué. Le résultat souhaité est que l'ensemble de la plateforme puisse être vue comme un macro-processeur dédié aux tâches d'acquisition et traitement d'images.

Chapitre 7 - Résultats expérimentaux

Différentes applications ont été implémentées, visant à démontrer l'efficacité de la plateforme SeeMOS, ainsi que de la méthodologie de développement proposée. Ces applications seront expliquées et leur implémentation détaillée. Les résultats expérimentaux obtenus seront analysés et commentés. Les exemples d'application sont le mode caméra rapide à 1000 images par seconde, l'acquisition synchronisée de données image et inertielles, la détection de mouvements par différence d'images et le tracking d'un motif par corrélation.

Chapitre 8 - Conclusion

Ce dernier chapitre viendra clore cette thèse, en soulignant les aspects originels qui ont été exploités et les possibles retombées de ces travaux, ainsi que les perspectives de travaux futurs.

Chapitre 2

Motivations et Problématique

"Confidence is what you have before you understand the problem. "

"La confiance est ce que l'on a avant de comprendre le problème."

Woody Allen, cinéaste, acteur, musicien et écrivain nord-américain.

Il est connu qu'une des plus grandes difficultés de la vision par ordinateur est liée à la grande quantité de données constituant une image. De nos jours, avec les imageurs de haute résolution, ce problème est accentué. Même si les interfaces de communication et dispositifs de traitement de l'information ont, eux aussi, profité d'un développement technologique important, il y a toujours un goulot d'étranglement provoqué par ce volume important de données à traiter et à communiquer.

La vision active propose, entre autres, de contourner cet inconvénient en s'appuyant sur une approche algorithmique guidée par la tâche. En effet, la grande quantité de données (pixels) d'une image ne constitue pas forcément une grande quantité d'informations sur la scène observée. En général, l'information sur la scène, et plus précisément l'information pertinente pour répondre à une question donnée, est concentrée seulement sur quelques régions de l'image. En conséquence, selon la tâche à exécuter, le système peut concentrer ses efforts de calcul uniquement sur ces régions, résultant ainsi dans une forte réduction de la quantité de données à manipuler.

Mais, si nous supposons que le système n'a pas de connaissance à priori sur l'environnement observé, par quelle moyen pourra-t'il connaître la position dans la scène, et donc dans l'image, de l'information qu'il recherche?

Pour essayer de répondre à cette question, nous allons d'abord analyser le comportement du système visuel humain. L'être humain est capable de réaliser une quantité innombrable de tâches visuelles, parmi elles la navigation en environnement inconnu et non structuré, l'évitement d'obstacles fixes ou en mouvement, la reconnaissance de formes, le suivi d'objets, la détection de mouvement et bien d'autres. Toutes ces tâches font l'objet d'études de la part de la communauté vision par ordinateur. Elles sont essentielles dans le contexte des systèmes mobiles autonomes, de la robotique industrielle et de la vidéo-surveillance, pour ne citer que quelques exemples. Il s'avère que ces tâches que nous réalisons quotidiennement, de façon automatique et très efficace, peuvent se montrer très complexes d'un point de vue algorithmique, et leur implémentation dans un système de vision artificielle représente un défi scientifique considérable. Il semble donc pertinent de chercher dans la vision humaine (ou biologique en général) quelques indices, idées et stratégies qui peuvent être précieuses afin de relever le défi de la vision artificielle.

Parmi les caractéristiques essentielles de la vision humaine, nous retiendrons particulièrement les suivantes :

- Les yeux sont des capteurs actifs, s'adaptant dynamiquement à l'environnement et au contexte (fig. 2.1 [32]);
- La résolution de la rétine est de type fovéale (décroissante en fonction de la distance par rapport à l'axe optique) (fig. 2.2 [33]);
- Il existe dans le cerveau des structures spécialisées dans le traitement "bas-niveau" de l'image (notamment le cortex visuel, fig. 2.3 [32]).

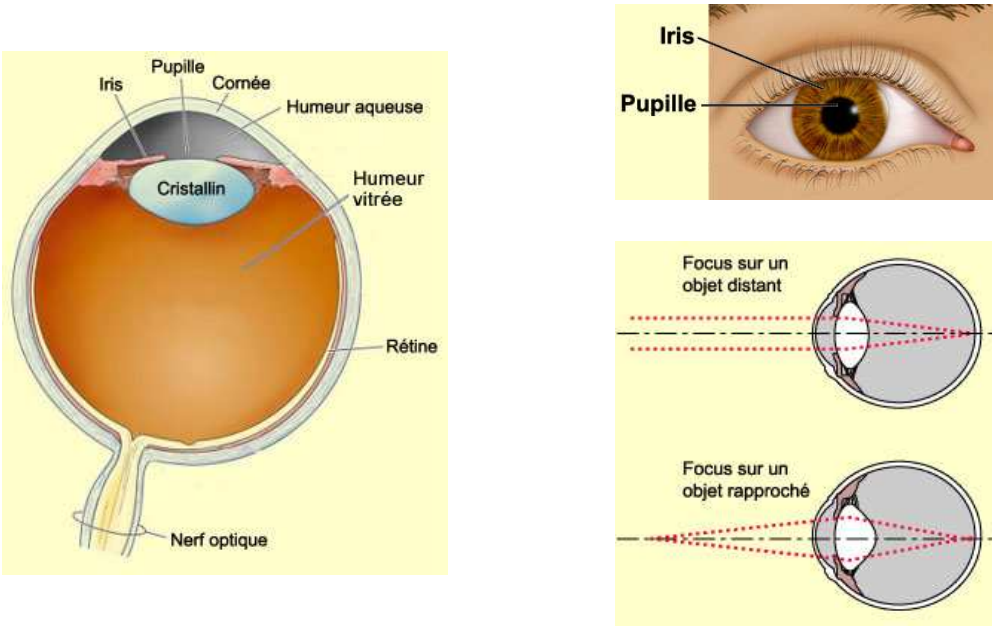


FIG. 2.1 – *Schémas descriptifs de l'oeil humain.*

Dans le système visuel humain, les yeux sont les responsables directs de l'exécution de plusieurs tâches, parmi elles le contrôle de l'ouverture de la pupille en fonction de l'éclairement et la déformation du cristallin, afin de réaliser la mise au point des images dans la rétine (fig. 2.1). Les mouvements des yeux sont aussi d'une extrême importance, car ils permettent de placer la région d'intérêt dans l'image, selon la tâche à exécuter, dans la zone de haute résolution de la rétine, appelée fovéa. Ces aspects caractérisent les yeux comme des capteurs actifs, par leur capacité d'adaptation autonome et dynamique aux stimuli provenant de l'environnement.

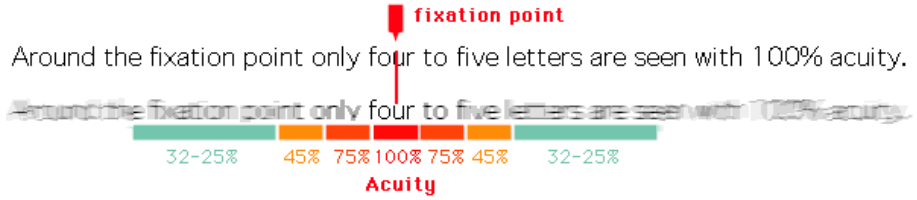


FIG. 2.2 – *Exemple de la résolution fovéale de la vision humaine lors de la lecture.*

La figure 2.2 illustre l'hétérogénéité de la résolution spatiale de la rétine. En effet, la rétine humaine est constituée d'une zone centrale de haute-résolution, appelée fovéa, entourée de zones concentriques dont la résolution est décroissante en fonction de l'éloignement par rapport à la fovéa. Si la fovéa est extrêmement importante afin d'apercevoir "en détail" certaines zones de l'image, la zone périphérique de basse résolution permet d'obtenir une grande quantité d'informations qualitatives

sur la scène, notamment la présence de mouvement ou d'objets et zones "saillantes", i.e. présentant des caractéristiques particulières par rapport à leur entourage (un panneau de couleur rouge placé au milieu d'un champ de tournesol par exemple).

Cette résolution hétérogène, combinée avec le mouvement contrôlé des yeux, permet à l'être humain de conserver un champ de vision large, tout en étant capable d'apercevoir des détails fins sur certaines zones, et en conservant une réactivité élevée par rapport aux événements de la scène, même quand ceux-ci ont lieu en dehors de la zone principale de focalisation (fovéa).

Toutes ces actions ont lieu simultanément, sans pour autant "surcharger" le système central de traitement (cerveau), car nous sommes parfaitement capables de voir et observer sans déployer un effort particulier (et conscient) pour cela.

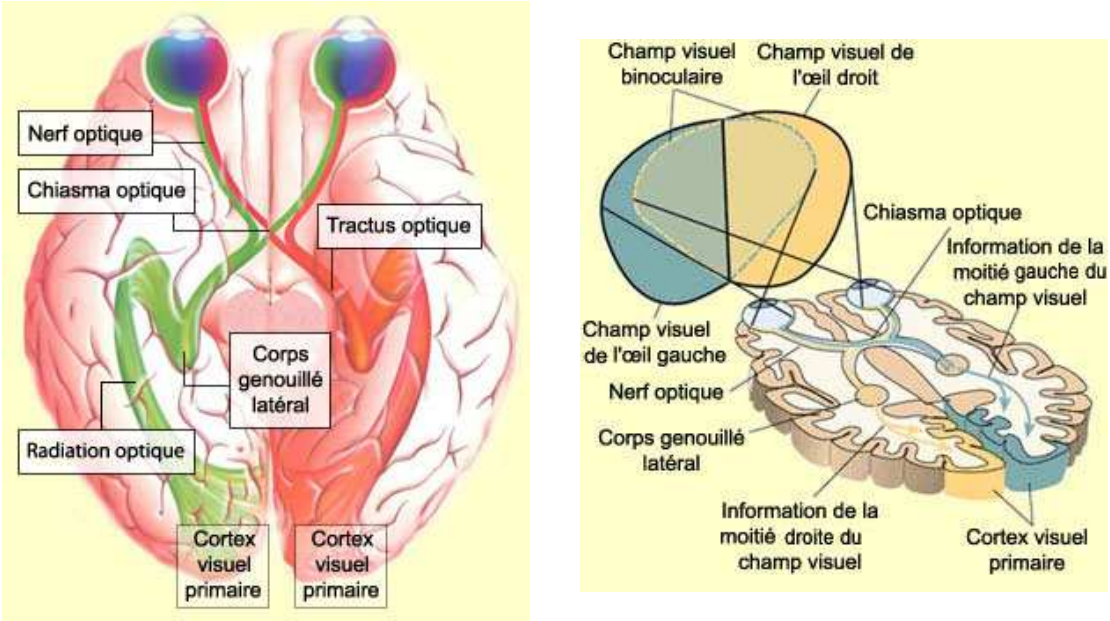


FIG. 2.3 – *Schémas illustrant la connexion entre les yeux et le cortex visuel par le nerf optique.*

La figure 2.3 illustre le chemin parcouru par l'information visuelle acquise par les yeux. Le nerf optique réalise la transmission de ces informations jusqu'au cortex visuel, en passant par les "Corps Genouillés Latéraux" (CGL). Le cortex visuel, ainsi que les CGL, sont des structures du cerveau spécialisées dans le traitement de l'information visuelle. Il est important de remarquer que cette première couche de traitements sur l'information acquise a lieu avant l'arrivée de l'information dans les zones du cerveau responsables par la prise consciente de décisions, l'identification et la reconnaissance.

Ce pré-traitement de l'information est exécuté de façon autonome et pratiquement inconsciente. Dans la suite de ce chapitre nous nous intéresserons à ces pro-

cessus visuels, que nous appellerons la vision précoce. Notre objectif est d'analyser et extraire les caractéristiques essentielles de la vision précoce, afin de les adapter dans le cadre d'un système de vision artificielle.

2.1 Vision précoce

Nous admettons qu'en général les processus visuels peuvent être décomposés en trois étapes distinctes : attention, focalisation, et haut-niveau. Ces trois étapes sont décrites ci-dessous :

Attention Les étapes d'attention servent à détecter des zones de l'image pouvant contenir des informations potentiellement intéressantes. Le processus d'attention visuelle peut être guidée par une caractéristique précise recherchée (par exemple la présence de mouvement dans la scène), ou peut être simplement associée à la détection de zones présentant des caractéristiques singulières qui les différencient de leur voisinage proche (couleur, contraste, orientation, etc.). Des exemples de tâches d'attention sont la détection de mouvement et la construction de cartes de saillance (section 2.1.1).

Focalisation La focalisation consiste à concentrer les efforts d'acquisition et traitement sur une zone précise de l'image. La focalisation peut être alimentée par un processus d'attention ayant détectée une zone "saillante", ou peut être guidée par le module haut-niveau qui recherche une information provenant d'une région précise de la scène. Le processus de vision par saccades du système visuel humain illustre l'étape de focalisation(section 2.1.2).

Haut-niveau Le module haut-niveau est le responsable de l'interprétation de la scène observée (identification, classification ou prise de décision). Ce module peut contrôler les processus d'attention et focalisation afin d'obtenir les données ou primitives les plus pertinentes pour la réalisation d'une tâche donnée.

Les étapes d'attention et focalisation sont responsables de la sélection et de l'acquisition des données, afin d'extraire l'information visuelle qui sera ensuite traitée par le module haut-niveau. Ces deux étapes fonctionnent comme des modules de pré-traitement, et sont communément appelées vision précoce (fig. 2.4).

Ces tâches de pré-traitement peuvent être liées à l'acquisition sélective d'une zone de l'image, à l'abstraction de primitives à partir des données acquises, ou au conditionnement de ces données. Ainsi, des processus tels que l'optimisation de contraste [34] ou le filtrage du bruit peuvent être considérées comme des tâches de vision précoce, tout comme le fenêtrage, la détection de mouvement [35] ou le suivi d'un objet [36]. Dans le système visuel humain, ces tâches sont réalisées par les yeux, au moyen de ses mouvements (saccades), de la dilatation des pupilles, de l'action du cristallin, etc. La combinaison de ces comportements, suivant une stratégie guidée

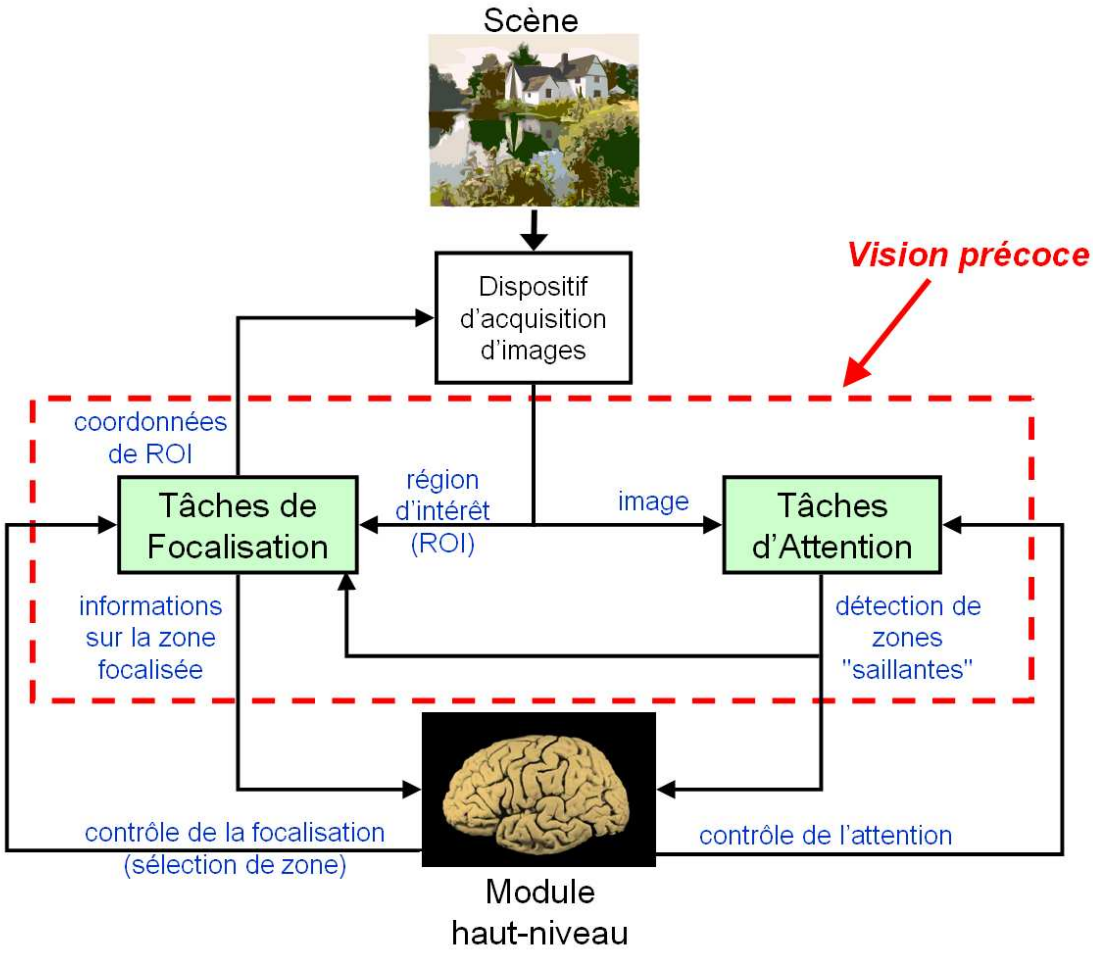


FIG. 2.4 – *Représentation des tâches de vision précoce au sein d'un système de vision.*

par la tâche, permet l'acquisition d'un ensemble d'informations pertinent afin de résoudre un problème donné, comme par exemple la lecture d'un panneau ou la reconnaissance d'un visage.

2.1.1 Attention visuelle et cartes de saillance

Les processus d'attention visuelle ont été le sujet de plusieurs travaux académiques, aussi bien de la part de la communauté des neurosciences [37] que de la communauté vision par ordinateur [38]. Tous ces travaux mettent en évidence l'existence du phénomène dit de "popping-out". Le "pop-out" désigne le fait que certaines régions d'une scène nous "sautent aux yeux". Ces phénomènes permettent au système de vision de diriger le regard en priorité vers les zones de l'image susceptibles de contenir des informations importantes.

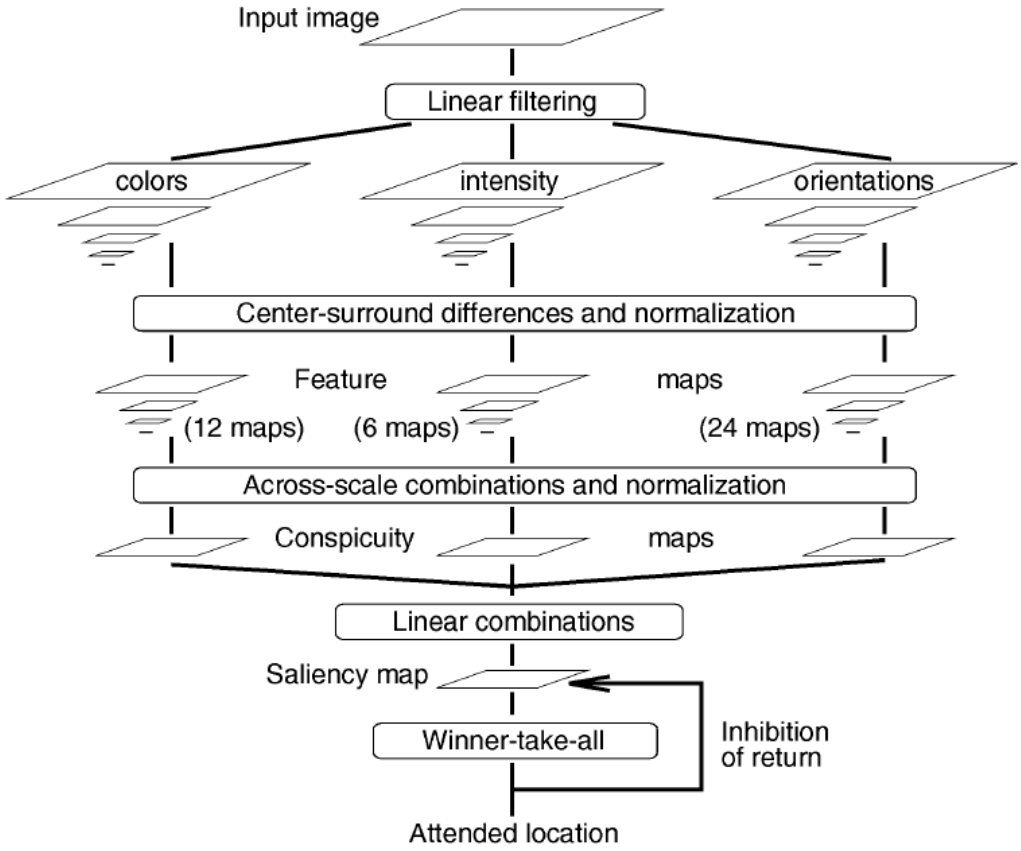


FIG. 2.5 – *Modèle général de Carte de Saillance proposé par Itti, Koch et Niebur*

Des modèles d'architecture bio-inspirées ont été proposés pour adapter le phénomène de "popping-out" aux systèmes artificiels. L'architecture proposée par Itti, Koch et Niebur [39] est illustrée en figure 2.5.

Ce modèle est basé sur la construction de cartes multi-résolution représentant différentes caractéristiques élémentaires de l'image (couleur, intensité et orientations). Ces cartes sont traitées afin de faire apparaître les points singuliers (*conspicuous*), et ensuite les différentes résolutions sont combinées. Finalement, les cartes des différentes caractéristiques sont pondérées et combinées entre elles, pour obtenir en résultat la "carte de saillance" de l'image (*saliency map*).

Le mouvement peut aussi être intégré en tant que caractéristique élémentaire [40]. En effet, le mouvement (gradient temporel de la luminosité dans une image) est une des primitives visuelles les plus saillantes. Dés qu'un objet mouvant entre dans notre champ de vision, notre attention est immédiatement attirée par cet objet.

A partir d'une carte de saillance, le système peut déterminer quelles régions de

l'image méritent d'être focalisées en priorité. Un capteur d'images intelligent avec des modules d'attention intégrés est présenté dans [41].

2.1.2 Vision par saccades

Les mouvements par saccades des yeux sont extrêmement importants en raison de la résolution hétérogène de la rétine. Comme il a déjà été illustré (fig. 2.2), la résolution spatiale de la rétine décroît en fonction de l'excentricité par rapport à l'axe optique. Les mouvements saccadés des yeux permettent donc de fixer la zone centrale de haute résolution (fovéa) sur des points particuliers de la scène, c'est à dire les points présentant des caractéristiques "saillantes" par rapport à leur voisinage. La zone périphérique de la rétine reste concentrée sur le restant de la scène, afin d'identifier des nouveaux points "saillants" candidats aux prochaines fixations de la fovéa.

Dans le cerveau, les images sont construites à partir d'une combinaison de plusieurs actions de saccade/fixation. Ce comportement peut être résumé par des tâches d'attention et focalisation exécutées en parallèle, et avec différentes résolutions spatiales, voire temporelles. Cette stratégie permet d'acquérir des données de façon sélective, visant à alimenter le système haut-niveau d'informations en accord avec son état et son objectif.

Dans la figure 2.6 nous avons plusieurs exemples du mouvement saccadé des yeux suivant une stratégie de recherche guidée par la tâche. Ce comportement a été mis en évidence par le psychologue russe Alfred L. Yarbus, et publiée en Russie en 1965, et aux États-Unis en 1967 [42].

L'image en haut à gauche est une reproduction du tableau de Ilya Repin[1] "An Unexpected Visitor". Un individu est appelé à examiner l'image afin de répondre à une certaine question. Les images suivantes représentent sept enregistrements différents des mouvement des yeux de l'individu, quand on lui demande d'exécuter les tâches suivantes :

1. Observation libre.
2. Estimer les moyens matériels de la famille.
3. Donner l'age des personnages.
4. Déduire ce que faisait la famille avant l'arrivée du visiteur inattendu.
5. Mémoriser les habits portés par les personnages.
6. Mémoriser la localisation des personnes et des objets dans la scène.
7. Estimer combien de temps le visiteur inattendu a été loin de la famille

1. Peintre et sculpteur russe des XIXe et XXe siècles (1844 - 1930)

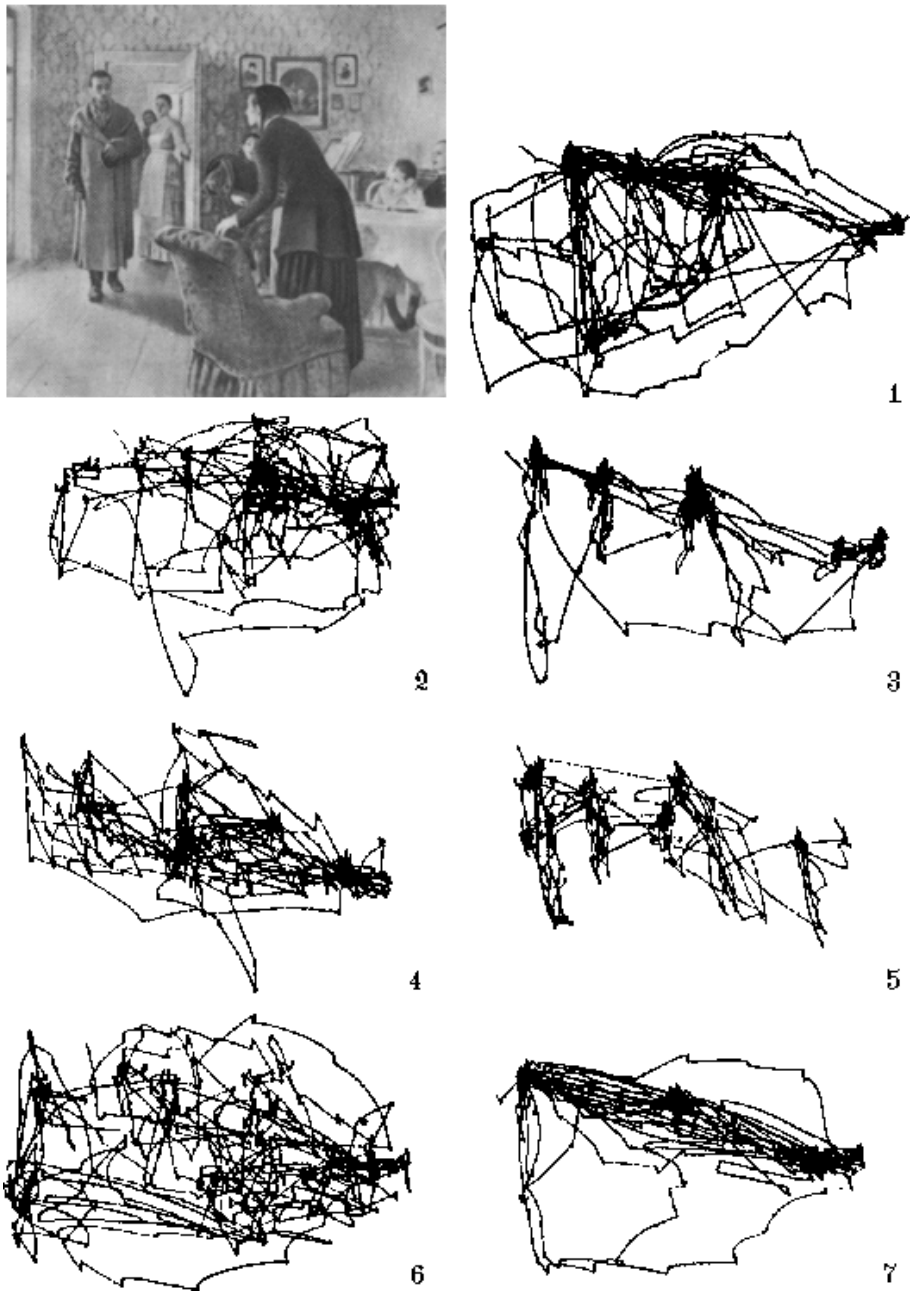


FIG. 2.6 – *Expérience de Yarbus mettant en évidence la stratégie exploratoire guidée par la tâche.*

L'enregistrement des mouvements des yeux démontre que la stratégie d'observation est fortement dépendante de la question posée à l'individu. Des travaux plus récents sur les mouvements saccadés des yeux peuvent être retrouvés dans [43].

La résolution hétérogène de la rétine combinée avec la stratégie de saccade/fixation résulte dans une réduction importante de la quantité de données acquises.

Ceci permet aux êtres humains d'avoir une bonne résolution d'image, un champ de vue large et une fréquence d'acquisition satisfaisante. Une approche similaire peut être appliquée au sein d'un système de vision artificielle, évitant ainsi les goulots d'étranglement provoqués par la transmission d'images haute résolution.

Dans un système électronique, les saccades "mécaniques" des yeux peuvent être simulés par une stratégie d'acquisition contrôlée, notamment au sein de capteurs de type CMOS, grâce à la capacité d'adressage aléatoire caractéristique de ces dispositifs. Les saccades se traduisent par le fenêtrage et l'acquisition de zones d'intérêt dans l'image.

Un exemple de système électronique d'acquisition par saccades est donnée dans [44]. Une architecture de capteur multi-résolution à fovéa adaptable est proposée dans [45].

2.1.3 Routines visuelles et détecteurs actifs

Shimon Ullman a proposé en 1984 [46] un modèle basé sur la décomposition des tâches visuelles en "opérations précoces" et "routines visuelles":

- Les opérations précoces sont des processus ascendants (*bottom-up*), exécutés en parallèle sur tout le contenu de l'image. La fonction de ces opérations est de créer une représentation de base de la scène, contenant des formes élémentaires (droites, splines, ...). Cette représentation s'assimile à la première ébauche (*primal sketch*) du paradigme de Marr (section 1.2 et figure 1.2). Il s'agit d'une représentation abstraite et non-liée de l'information visuelle.
- Les routines visuelles sont des processus descendants (*top-down*), basés sur l'application séquentielle d'opérateurs basiques de vision sur une région d'intérêt. Leur fonction est d'extraire les propriétés et relations spatiales des éléments constituant la représentation de base. L'objectif est de lier ces éléments afin de définir des objets ou parties d'objets.

Les routines visuelles sont donc des processus cognitifs, assemblés à partir d'un nombre fixe d'opérations élémentaires. Ces routines viennent extraire des informations à partir d'une représentation de base. Les résultats obtenus sont inscrits dans une représentation incrémentale, qui pourra être utilisée et incrémentée par les routines suivantes.

Ullman défend que des routines sophistiquées, capables de percevoir la forme et la relation spatiale des parties d'un objet, peuvent être construites à partir des 5 opérations suivantes :

1. Changement du focus de traitement (*Shifting the processing focus*): permet à une routine visuelle d'être appliquée à une partie définie du champ de vision.

Cette opération est similaire aux processus de focalisation et aux saccades de l'oeil.

2. Sélection des zones à traiter (*Indexing*): il s'agit de sélectionner les zones de l'image présentant des caractéristiques singulières et pouvant donc contenir des informations intéressantes. Similaire au phénomène de "pop-out" et au concept de saliency map.
3. Activation délimité (*Bounded activation ou coloring*) : Consiste à propager un signal d'activation ("couleur") sur une surface de la représentation de base, à partir d'un point ou contour donné. L'activation s'arrête dans les frontières (discontinuités) de cette surface. Ceci sert à faire ressortir une certaine zone par rapport au *background* afin de faciliter son identification.
4. Extraction de contour (*Boundary tracing*) : similaire à l'activation délimitée, mais appliquée à un contour plutôt qu'à une surface. Cette routine permet de déterminer si deux points se trouvent sur un même contour, ou sur les frontières d'un même objet.
5. Marquage (*Marking*) : sert à indiquer si une certaine région à déjà été traitée. Permet de garder une trace des routines appliquées pour une question de contrôle et coordination.

Dans sa thèse, Pierre Chalimbaud [31] propose l'introduction d'un aspect rétro-actif dans l'approche modulaire proposée par Ullman. Car même si cette dernière est à la fois ascendante et descendante (*bottom-up* et *top-down*), il en demeure que les opérations élémentaires présentées ci-dessus sont purement passives. Chalimbaud propose donc l'utilisation de "détecteurs actifs".

Un détecteur actif est constitué en associant une zone d'intérêt dans l'image, une solution algorithmique dont l'objectif est d'extraire une information locale, et une architecture d'implantation dédiée (fig. 2.7).

Par rapport aux routines visuelles de Ullman, un détecteur actif jouit de plus d'autonomie et d'un contrôle direct des paramètres d'acquisition d'image. La rétro-action s'exprime à trois niveaux :

- niveau local, par l'adaptation de son procédé de détection en fonction des résultats obtenus ;
- niveau sensoriel, par le paramétrage de l'acquisition et l'adaptation de sa zone d'intérêt dans l'image ;
- niveau supervisé, par un système superviseur qui peut contrôler et paramétrer le détecteur actif en fonction de la tâche à exécuter et de la stratégie adoptée.

Les détecteurs actifs peuvent être vus comme des opérateurs autonomes de vision précoce.

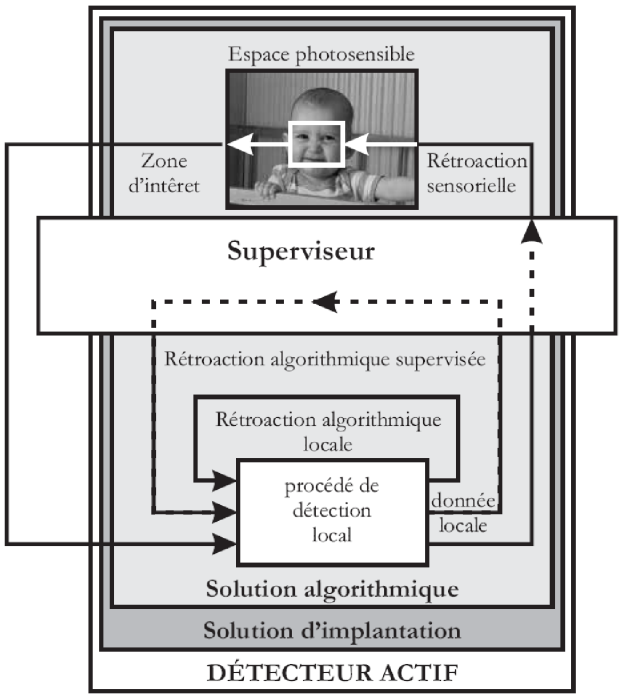


FIG. 2.7 – *Schéma synoptique d'un détecteur actif.*

Dans cette section furent présentées et détaillés différents aspects et facettes de la vision précoce. Ces notions nous serviront de base pour extraire dans la section suivante les contraintes matérielles et opérationnelles imposées par ce cadre applicatif et conceptuel.

2.2 Contraintes matérielles et opérationnelles de la vision précoce

Les nouvelles technologies d'intégration sur silicium permettent la création de capteurs avec des résolutions de plus en plus élevées (de l'ordre de quelques millions de pixels par image, voire une dizaine de Mpixels pour les appareils photos les plus récents). En conséquence, les cadences d'acquisition connaissent elles aussi une augmentation importante (l'imageur CMOS MT9M413 de chez Aptina Imaging, ancienne Micron Imaging, est capable d'acquérir jusqu'à 660 Mpixels/s). Une exploitation adéquate des capacités de tels capteurs impose des contraintes temporelles très strictes au système de traitement de l'information, ainsi qu'à l'interface de communication le reliant au module d'acquisition.

La vision active, et particulièrement la vision précoce, permettent de réduire

la quantité de données à traiter et transmettre par le contrôle sélectif du processus d'acquisition. Ceci viendra alléger les contraintes du module de communication, ainsi que la charge de calcul du module haut-niveau. Cette diminution quantitative des données doit être accompagné d'une amélioration qualitative. En quelques mots, l'objectif est de transmettre moins de "données", mais plus "d'informations".

Par contre, même si une approche active, quand comparée à l'approche passive classique, a tendance à simplifier les tâches de vision haut-niveau, elle crée des nouvelles exigences au niveau de l'exécution des tâches de vision précoce. Celles-ci demeurent relativement gourmandes en ressources de calcul, et ont des exigences opérationnelles spécifiques. Une architecture matérielle dédiée à la vision précoce doit prendre ces aspects en considération, et offrir une plateforme adaptée à l'exécution de telles applications.

Un système de vision précoce est sensé agir au sein même du module d'acquisition (capteur), afin de "sélectionner" les données (pixels) à acquérir, et contrôler les paramètres d'acquisition. Ce processus est notamment illustré par la stratégie de vision par saccades. La sélection et le contrôle sont directement liés à la tâche à exécuter.

Ce comportement guidé par la tâche impose les premières spécificités du cadre d'application : d'une part il doit exister une rétroaction du module de traitement sur le capteur, afin de permettre l'adaptation des paramètres d'acquisition en fonction de la tâche. D'autre part, le contrôle du capteur doit pouvoir être réalisé de façon dynamique et rapide, afin de satisfaire les contraintes temporelles et assurer la réactivité du système. Ceci suppose donc un module de traitement proche du capteur, afin de minimiser le surcoût (*"overhead"*) dû à la communication.

Cette proximité entre le module de traitement et celui d'acquisition, ainsi qu'un lien "privilégié" de contrôle entre les deux, suggère l'utilisation d'un système de traitement embarqué au sein même de la caméra.

Une autre spécificité des systèmes de vision précoce provient de la nature des traitements réalisés, et de leurs différentes granularités. Les tâches d'attention et focalisation peuvent à ce titre être décomposées essentiellement en deux catégories de traitements :

Les traitements de bas-niveau (granularité pixel) Ces traitements consistent à appliquer une ou plusieurs opérations simples à chaque pixel, ou à un groupe de pixels adjacents. Généralement ces traitements sont répétés un grand nombre de fois à l'intérieur d'une même image ou fenêtre d'intérêt.

Les traitements de moyen-niveau (granularité image) Ces traitements sont appliqués au niveau d'une image ou fenêtre, et sont répétés pour chaque image d'une séquence. L'objectif est d'extraire des primitives ou indices pouvant décrire la scène observée, ou servant à contrôler le processus d'acquisition.

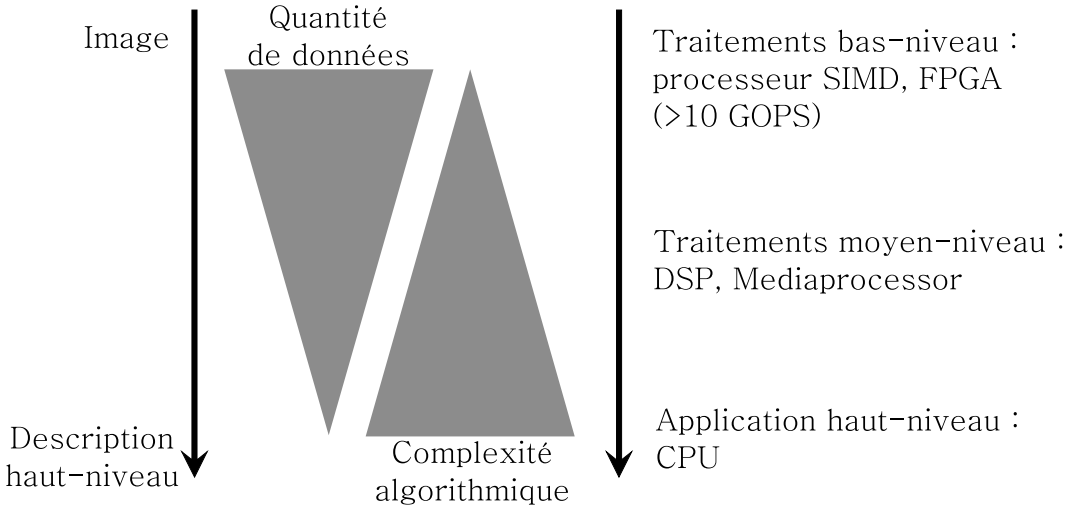


FIG. 2.8 – *Quantité de données vs. Complexité algorithmique* (GOPS = Giga (10^9) Opérations par Seconde)

La figure 2.8 illustre la relation inverse entre la quantité de données à traiter et la complexité des algorithmes.

Les traitements bas-niveau réalisent quelques opérations simples, qui sont répétées sur un très grand nombre de données. Ils mettent fréquemment en oeuvre des opérations dites de "voisinage". Des exemples sont la convolution par un masque constant (calcul du gradient), ou le calcul de la corrélation entre deux groupes de pixels (appariement de primitives). La répétition de ces opérations de voisinage sur chaque pixel d'une image implique un très grand nombre de transactions mémoire, et peuvent rapidement devenir un goulot d'étranglement si l'architecture matérielle du système n'est pas adaptée à ce type de traitement.

L'exploitation du parallélisme potentiel (tâches et données) est alors une opportunité, voire une nécessité. Ces traitements devant impérativement être réalisés en temps-réel, la parallélisation peut en effet aider à respecter les exigences temporelles. D'autre part, certaines méthodes de la vision précoce présentent des caractéristiques intrinsèquement parallèles. Celles-ci supposent fréquemment l'exécution concurrente de différentes tâches sur un même jeu de données. Un exemple est la construction des cartes multi-résolution de couleur, contraste et orientation pour la création d'une carte de saillance.

Les processus de bas-niveau étant fortement parallélisables, ils sont donc adaptés aux structures de calcul telles que les composants FPGA ou les processeurs SIMD (Single Instruction Multiple Data).

D'un autre côté, les traitements de moyen-niveau (comme la segmentation ou l'estimation de déplacement pour suivi d'objet) peuvent employer des routines mathéma-

tiques complexes, itératives ou récursives, comme l'inversion de matrices ou les méthodes de minimisation. Ces routines séquentielles, comptant un grand nombre d'instructions et opérations, sont appliquées sur un nombre restreint de descripteurs ou primitives.

Les architectures classiques du type PC conviennent à ce type de traitement, plus par leur haute cadence de fonctionnement (de l'ordre du GHz) et par leur excellente programmabilité que par leur caractéristiques architecturales. Par contre, dans un contexte embarqué, où les fréquences d'horloge sont nettement inférieures, l'exécution efficace de ce genre de routine constitue un défi considérable. Des structures matérielles dédiées au traitement du signal et la possibilité de programmer ces routines en langage haut-niveau sont alors des caractéristiques souhaitables. Ces processus sont donc adaptés aux structures basées sur un coeur de CPU, comme les DSP's ou les *mediaprocessors* (dispositifs dédiés au traitement temps-réel des flux (*streaming*) audio ou vidéo).

La nécessité de structures matérielles adaptées aux types de traitement vient une fois de plus renforcer l'idée de l'utilisation d'un dispositif dédié unique, concentrant acquisition et traitement. Ces dispositifs sont connus sous le nom de "caméras intelligentes", ou *"Smart Cameras"* selon le terme consacré en anglais.

2.3 Caméras Intelligentes, ou *Smart Cameras*

Les systèmes embarqués connaissent un intérêt grandissant des communautés scientifique et industrielle. Les avancés technologiques permettent aujourd'hui la conception de systèmes de plus en plus complexes et complets au sein d'un dispositif unique. Les caméras intelligentes, ou *Smart Cameras*, font partie de ce processus d'évolution [47, 48].

Une caméra intelligente peut être grossièrement définie comme un système de vision comportant non seulement des dispositifs d'acquisition d'images (capteur) et de communication (interface), mais aussi des éléments capables de traiter les données acquises au sein même de la caméra.

Par contre, cette définition inclut la plupart des caméscopes et appareils photos récents, car ceux-ci comportent souvent des structures de traitement permettant d'améliorer le rendu des images. Ce ne sont pas pour autant des caméras intelligentes, car leur objectif est strictement "esthétique", c.à.d. d'améliorer la qualité des images pour une visualisation postérieure.

Nous ajouterons donc à la définition de "caméra intelligente" le fait que les traitements réalisés ont pour objectif l'extraction d'informations sur la scène observée, pour une application autre que le simple rendu de l'image [49].

L'utilisation de ressources de calcul embarquées permet à ce genre de dispositif de s'adapter à un grand nombre d'applications, présentant des avantages telles que :

- Dispositifs mobiles et autonomes : dispense de l'utilisation d'un système hôte ("mainframe"), permettant l'obtention de dispositifs de taille réduite et de basse consommation d'énergie (pour les UAV's (Unmanned Aerial Vehicle) par exemple).

- Perception active : pour ces applications, la rétroaction entre traitement et perception est fondamentale (section 2.2). Cela implique un lien privé direct entre le capteur et l'unité de traitement, impossible à obtenir avec les bus classiques de type USB, Firewire, Ethernet, etc.

- Réseaux de caméras : dans des telles configurations, le traitement distribué des informations par des caméras intelligentes présente des avantages majeurs par rapport à un système de traitement centralisé [50]. D'une part la quantité de données sensorielles à transmettre est fortement diminuée, évitant les goulots d'étranglement qui sont d'autant plus problématiques quand plusieurs caméras sont en jeu. D'autre part, grâce à l'autonomie de tels dispositifs, l'envoi d'informations de contrôle vers les caméras est réduit, ou même annulé.

Les caméras intelligentes peuvent en effet être déclinées en deux modes : comme un capteur intelligent, connecté à un système hôte, ou comme un dispositif indépendant et autonome (*stand-alone*) capable d'analyser une scène et déclencher des événements externes si une certaine condition est retrouvée.

Les chapitres suivants sont consacrés à l'étude des caméras intelligentes, d'abord d'un point de vue matériel, et puis du point de vue de l'implémentation des applications sur ces plateformes.

Chapitre 3

État de l'art : Smart Cameras

"Il n'y a qu'une méthode pour inventer, qui est d'imiter. Il n'y a qu'une méthode pour bien penser, qui est de continuer quelque pensée ancienne et éprouvée."

Émile-Auguste Chartier, dit Alain, professeur, essayiste et philosophe français (1868 - 1951). Propos sur l'éducation (1932).

Les systèmes embarquées, tels que les caméras intelligentes, peuvent avoir des besoins architecturaux, physiques et opérationnels spécifiques. Consommation d'énergie, limitation de taille, opération temps-réel et autonomie peuvent apparaître comme étant des contraintes fortes, rendant le design de ces systèmes plus complexe quand comparé aux systèmes de type "desktop".

Heureusement, grâce à l'évolution observée ces dernières années dans le domaine de la microélectronique et des technologies VLSI (Very Large Scale Integration), une palette variée de dispositifs est disponible aujourd'hui pour la conception des systèmes embarqués de vision. Chaque famille de dispositifs présente ses avantages et inconvénients propres, faisant du choix des composants matériels un facteur prépondérant pour la performance, flexibilité et programmabilité d'un système donné. Le plus souvent, ces composants vont déterminer la capacité de ce système à héberger une application donnée, ou une classe d'applications.

Dans ce contexte, l'étude des composants matériels s'avère primordiale, aussi bien pour la conception de l'architecture d'un système que pour la mise en oeuvre de celle-ci. Souvent habitués aux langages de programmation haut-niveau, nous oublions parfois qu'à un certain moment les innombrables et abstraites lignes de code d'un programme sont transformées en signaux électriques bien réels, et que c'est effectivement à ce moment là, dans le silicium d'un composant, que le traitement de l'information prend forme. Il est donc essentiel de connaître les particularités et contraintes des dispositifs utilisés, afin de mieux prévoir et optimiser le comportement du système en fonction de l'application envisagée.

De façon générale, l'architecture matérielle d'une caméra intelligente peut être décomposée en trois modules principaux, comme illustré en figure 3.1 et décrit ci-dessous :

- Module d'acquisition de données : composé essentiellement d'un capteur d'images. Néanmoins, d'autres types de dispositifs sensoriels peuvent être intégrés, afin d'obtenir des informations supplémentaires sur la scène et l'environnement ;

- Module de traitement des données : l'exécution des différentes étapes du traitement de l'information est prise en charge ici. Les résultats obtenus peuvent être envoyés vers un système hôte, et/ou être utilisés pour contrôler l'acquisition de nouvelles données ;

- Module de communication : responsable de la connexion du système embarqué avec le monde extérieur (système hôte ou réseau).

Ce chapitre présente les principaux composants susceptibles d'intégrer chacun de ces modules, suivi d'exemples d'architectures issues aussi bien du milieu académique que du milieu industriel.

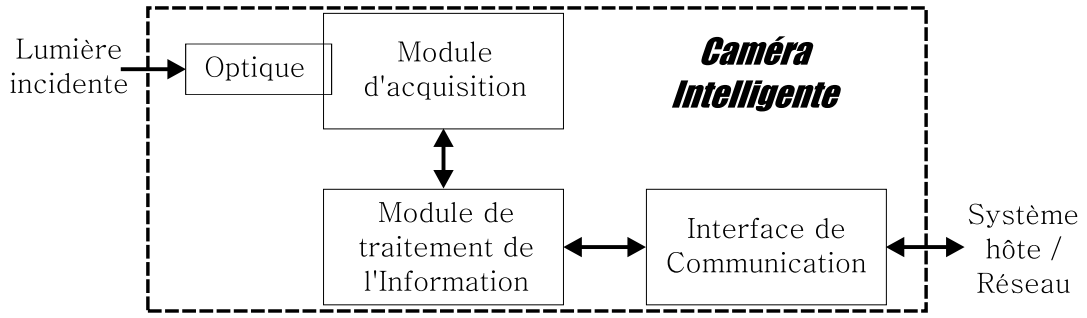


FIG. 3.1 – *Schéma synoptique simplifié de la structure interne d'une caméra intelligente.*

3.1 Composants et Technologies

Cette section dresse un panorama des composants et technologies déployés lors de la construction d'une caméra intelligente, en analysant leur potentiel et en décrivant leur principaux avantages et limitations [51]. Ces composants et technologies sont divisés en trois catégories, selon les trois modules fonctionnels décrits précédemment et illustrés en figure 3.1 : acquisition, traitement et communication.

3.1.1 Dispositifs d'acquisition de données

S'il est vrai que l'intégration d'un capteur d'images est une condition *sine qua non* pour la construction d'une caméra intelligente, le module d'acquisition de données ne se limite pas nécessairement à l'acquisition d'images uniquement. Ainsi, après la présentation des principales technologies d'imagerie existantes, d'autres dispositifs sensoriels pouvant intégrer le module d'acquisition seront analysés.

3.1.1.1 Capteurs d'images

S'agissant d'une caméra, il paraît évident que le composant d'acquisition d'images jouera un rôle central dans la performance du système. Même si la qualité des images acquises (sensibilité, dynamique) et la résolution (nombre de pixels) sont des facteurs très importants pour caractériser un capteur d'images, d'autres caractéristiques doivent être soigneusement prises en considération. Cela inclut le *frame rate* (nombre d'images par seconde), les modes d'adressage (possibilité de sous-échantillonnage, adressage aléatoire), la facilité d'intégration et la logique nécessaire pour le contrôle de l'acquisition (i.e. seulement quelques signaux suffisent pour synchroniser l'opération, ou des dizaines de paramètres et triggers sont nécessaires ?).

Les technologies CCD et CMOS sont les deux plus couramment rencontrées aujourd'hui dans les capteurs d'images. Les capteurs CCD (Charge-Coupled Device,

ou dispositif à transfert de charge) sont basés sur une technique de lecture (*readout*) par registre à décalage. Cela veut dire que les charges électriques accumulées par une photodiode (pixel) sont transférées vers le pixel voisin et ainsi de suite, la dernière photodiode étant connectée aux circuits d'amplification et d'échantillonnage (fig. 3.2).

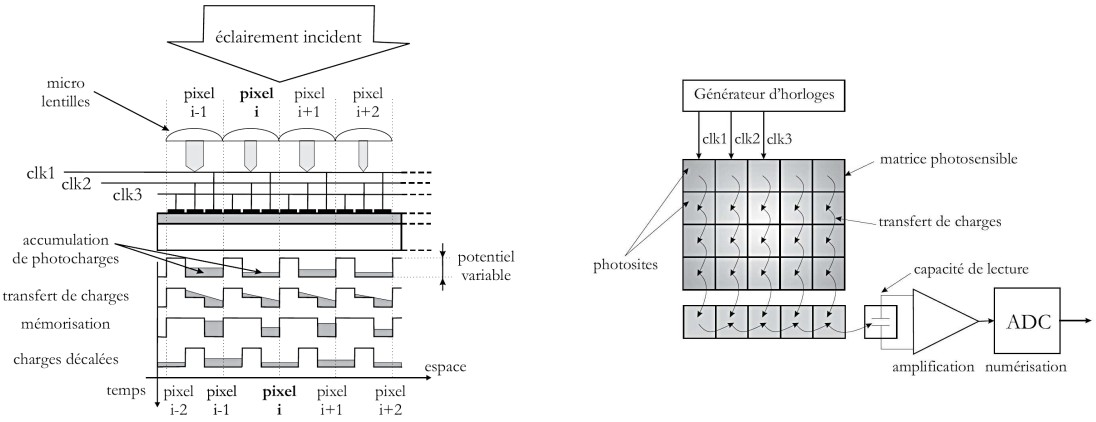


FIG. 3.2 – *Structure et technique de lecture (readout) d'un imageur CCD.*

Un circuit de contrôle permet de synchroniser le transfert des charges, jusqu'à que l'intégralité de la matrice photosensible soit lue. Dû à leur principe de construction, un problème reconnu des capteurs CCD a lieu quand la zone photosensible est surexposée. Si la capacité de stockage de charge maximale d'une photodiode est atteinte, les charges en excès peuvent "transborder" vers les photodiodes adjacentes, provocant un effet d'éblouissement (*blooming*), illustré en figure 3.3. Il existe des techniques anti-blooming, par l'incorporation de structures de drainage des charges.

Les principales avantages des capteurs CCD sont leur rapport signal/bruit élevé, une grande uniformité d'image et un haut facteur de remplissage (*fill factor*), permettant une bonne sensibilité à la lumière.

Les imageurs CMOS (Complementary Metal Oxide Semiconductor) adoptent une technique de lecture similaire à celle des mémoires RAM, avec des décodeurs de ligne et colonne (fig. 3.4). De cette façon, la lecture aléatoire des pixels de l'image devient possible, ce qui constitue un atout majeur de ce type de capteur.

La lecture sélective des pixels permet l'acquisition contrôlée de fenêtres d'intérêt de tailles, positions et formats variables. De ce fait, pendant que pour les capteurs CCD la cadence d'acquisition s'exprime en images par seconde (*fps - frames per second*), il est plus pertinent dans le cas des capteurs CMOS de parler en pixels par seconde. De cette façon, et contrairement aux capteurs CCD, la technologie CMOS permet l'obtention de *frame rates* très élevés, par l'adressage uniquement d'une petite portion de la matrice photosensible. Ceci s'avère très utile pour des applications de vision rapide, notamment en robotique et métrologie.

FIG. 3.3 – *Illustration du phénomène de blooming. Les lignes verticales sont provoqués par le débordement des charges vers les pixels voisins.*

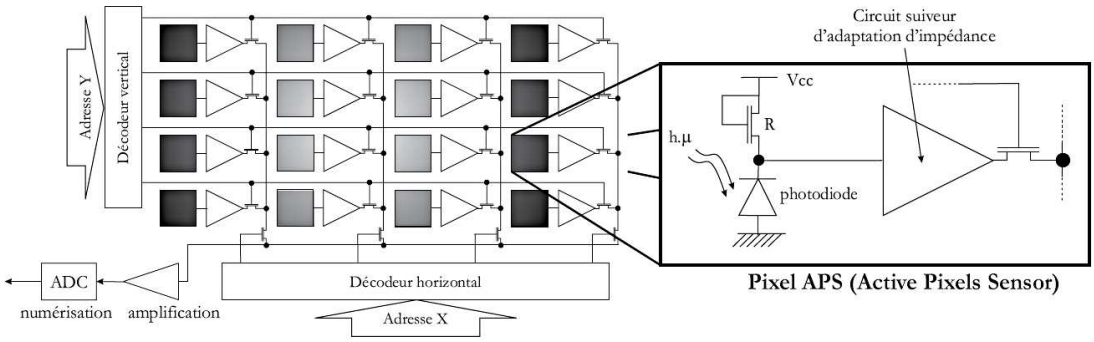


FIG. 3.4 – *Architecture d'un imageur CMOS.*

D'autres avantages des capteurs CMOS sont une dynamique élevée, leur simplicité d'intégration et peu ou pas de *blooming*. Par contre, malgré ces avantages, les capteurs CMOS présentent une qualité d'image inférieure quand comparés aux capteurs CCD, et particulièrement un plus grand FPN (Fixed Pattern Noise, ou bruit spatial fixe). Ce bruit est provoqué par la non-uniformité des caractéristiques électriques du circuit de chaque pixel, ainsi que des éléments du circuit de lecture, comme les amplificateurs. Ceci résulte dans une image finale contenant un *offset* statique différent pour chaque colonne. Ce bruit spatial fixe peut être facilement calculé et supprimé de chaque image, permettant l'obtention d'un résultat de meilleure qualité. Par contre, la suppression du FPN représente une charge supplémentaire pour le système de traitement.

En résumé, les technologies CCD et CMOS présentent chacune leurs avantages

et inconvénients, et aucune des deux technologies n'est dans l'absolu supérieure à l'autre. Le choix de la technologie la plus adaptée à une application se fera donc essentiellement sur le compromis entre la cadence et flexibilité de lecture (CMOS), ou la qualité de l'image (CCD). D'autres technologies d'imagerie existent, comme par exemple l'imagerie thermique et infrarouge, pour les applications de vision nocturne et de mesure non-destructive.

3.1.1.2 Autres dispositifs sensoriels

En plus du capteur d'images, une caméra intelligente peut être équipée d'autres types de capteurs, aussi bien proprioceptifs qu'extéroceptifs. Des accéléromètres, gyroscopes, microphones et modules GPS (Global Positioning System) peuvent être intégrés au sein du dispositif, fournissant des informations supplémentaires sur l'environnement et l'état de la caméra elle-même, pour des applications comme la stabilisation d'images, la reconstruction, la navigation, le contrôle d'un robot, etc.

Les capteurs inertiels (accéléromètres et gyroscopes) existent depuis plusieurs décennies, mais cette technologie a été longtemps réservée presque exclusivement à des applications coûteuses et de grande envergure, comme par exemple des projets militaires. Seulement dans les dernières années, avec l'avancement des techniques de fabrication et implantation sur silicium, ces capteurs sont devenus disponibles pour les dispositifs grand public.

Avec la réduction de leur taille et de leur prix, ces dispositifs sont devenus un choix logique pour les applications de perception proprioceptive du mouvement [52]. La démocratisation de cette technologie est notamment illustrée par sa présence dans la manette de la console de jeux Wii, de chez Nintendo, ou dans le iPhone de chez Apple.

La proprioception est en effet la capacité de mesurer des grandeurs physiques caractéristiques au système lui-même, comme son mouvement ou sa température. Dans le contexte d'une caméra en déplacement, les informations proprioceptives peuvent être utilisées par exemple pour compenser les mouvements de la caméra, dans des applications de suivi de motif ou de stabilisation d'images. En outre, la capacité d'estimer la position et l'orientation de la caméra peut permettre la réalisation d'un appariement stéréo sur une paire d'images acquises par une seule caméra [52].

D'autres dispositifs sensoriels sont disponibles aujourd'hui sous forme de modules OEM (Original Equipment Manufacturer), pouvant être intégrés au sein d'une caméra intelligente. Des modules GPS peuvent être utilisés pour évaluer la position absolue d'une caméra, faisant partie par exemple d'un système distribué de surveillance. La connaissance de la position peut permettre d'estimer l'orientation du soleil, et d'éliminer plus facilement certains problèmes dus à l'éclairement [53].

Des capteurs extéroceptifs tels que les microphones peuvent également être em-

ployés. Les événements sonores sont une source précieuse d'informations pour des tâches telles que la détection d'intrusion. En outre, un système stéréo de microphones peut donner des indices de localisation dans une application de suivi de personne [54]. Les microphones peuvent être intégrés dans le corps même de la caméra, ou comme des noeuds dans un réseau distribué de surveillance [55].

3.1.2 Dispositifs de traitement embarqué

Les dispositifs de traitement sont les structures responsables du traitement des données. L'architecture interne de ces éléments est composée d'opérateurs arithmétiques et logiques, registres, modules dédiés au contrôle et synchronisation des opérations, et éventuellement de mémoires internes afin de stocker les données et résultats. Ces structures peuvent être classifiés en trois familles :

- les éléments de type **fixe**, essentiellement des circuits ASIC dédiés à un traitement spécifique, aussi connus sous le nom d'accélérateurs matériels (multiplieurs, convolueurs, IP's (*Intellectual Property*), etc.) ;

- les éléments de type **reconfigurable**, notamment des circuits CPLD et FPGA, composés de nombreuses cellules logiques élémentaires qui peuvent être assemblées librement afin d'implémenter une fonction ;

- les éléments de type **programmable**, où un flot de contrôle (programme) détermine les opérations à exécuter, offrant une grande flexibilité d'application (processeurs généralistes, DSP, microcontrôleurs).

La figure 3.5 [56] donne un exemple de système faisant appel à des éléments de calcul de type fixe (accélérateur spécialisé), programmables (microprocesseurs, DSP, microcontrôleurs) et reconfigurables (opérateurs câblés (trait. #n) et IP's). On remarquera également la présence d'un coeur de processeur instancié dans le circuit FPGA. Ces dispositifs sont connus sous l'appellation de processeurs "soft-core", et consistent dans l'utilisation des ressources reconfigurables d'un circuit FPGA pour implémenter un dispositif programmable de traitement.

Dans l'exemple de la figure, tous les éléments sont reliés via un réseau d'interconnexion, permettant l'échange d'informations et données entre les différentes structures de traitement, ainsi que les mémoires et dispositifs d'entrée/sortie. Un deuxième bus est responsable du contrôle et configuration des éléments instanciés dans le FPGA. Il est intéressant de noter que d'autres réseaux d'interconnexion sont présents à l'intérieur même du circuit FPGA, ce qui démontre les différentes granularités pouvant coexister au sein d'un même système. Dû aux différentes natures des éléments de calculs, ce type d'architecture est appelée hétérogène.

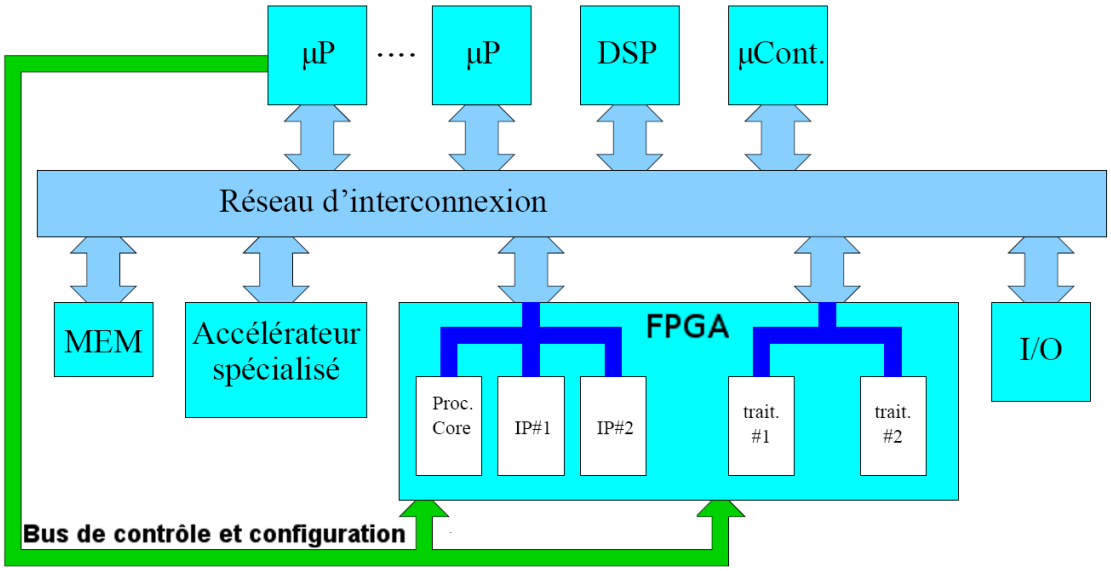


FIG. 3.5 – *Exemple de d'architecture hétérogène, faisant appel à différents types d'éléments de calcul interconnectés.*

Les caméras intelligentes peuvent être équipées de différents types de dispositifs de traitement. Les plus couramment employés sont les processeurs généralistes embarqués de type RISC (Reduced Instruction Set Computer), les microcontrôleurs [57, 58], les DSP [59], les circuits reconfigurables FPGA [60, 36] ou encore les processeurs dédiés au traitement de flux audio/vidéo (streaming), connus sous le nom de processeurs média, ou *mediaprocessors* [47]. D'autres dispositifs tels que les circuits ASIC (Application Specific Integrated Circuit), les processeurs SIMD [61] et les processeurs "soft-core" [35] peuvent être également exploités.

Ces dispositifs peuvent être intégrés dans des architectures hétérogènes (FPGA + DSP par exemple, comme la plateforme SeeMOS illustrée en section 3.3), ou des architectures multi-processeur [50], avec un réseau embarqué de plusieurs unités de traitement identiques (NoC - Network on Chip).

Le choix d'un dispositif de traitement et de l'architecture du système doit être fait en fonction des différentes contraintes posées par le cadre d'application. Ces contraintes peuvent être de nature physique, applicative, ou associées au design du système :

– Les contraintes physiques sont par exemple la taille du dispositif, la consommation d'énergie et le nombre de broches ou ports d'entrée/sortie.

– Les contraintes associées au design sont par exemple le prix du dispositif, le coût NRE (non-recurring engineering), la facilité d'intégration (packaging du dispositif, BGA, FBGA, DIP), ainsi que la circuiterie nécessaire pour le

fonctionnement du dispositif (résistances, capacités, alimentation, oscillateurs, etc.).

– Les contraintes applicatives sont liées à la puissance de calcul (e.g. le nombre d'instruction ou opération exécutées par seconde), à la programmabilité (e.g. langages haut-niveau, assembleur, langage de description matérielle), et à la flexibilité d'application.

Le plus souvent, un compromis doit être retrouvé parmi ces différentes caractéristiques et contraintes, selon le volume de production prospecté (prototype unique ou destiné au marché grand public?), et selon l'évolutivité souhaitée pour le système [62, 63].

La figure 3.6 présente une comparaison approximative entre les composants ASIC, DSP, FPGA et mediaprocessor, basée sur quelques unes des principales contraintes rencontrées lors du design d'un système embarqué.

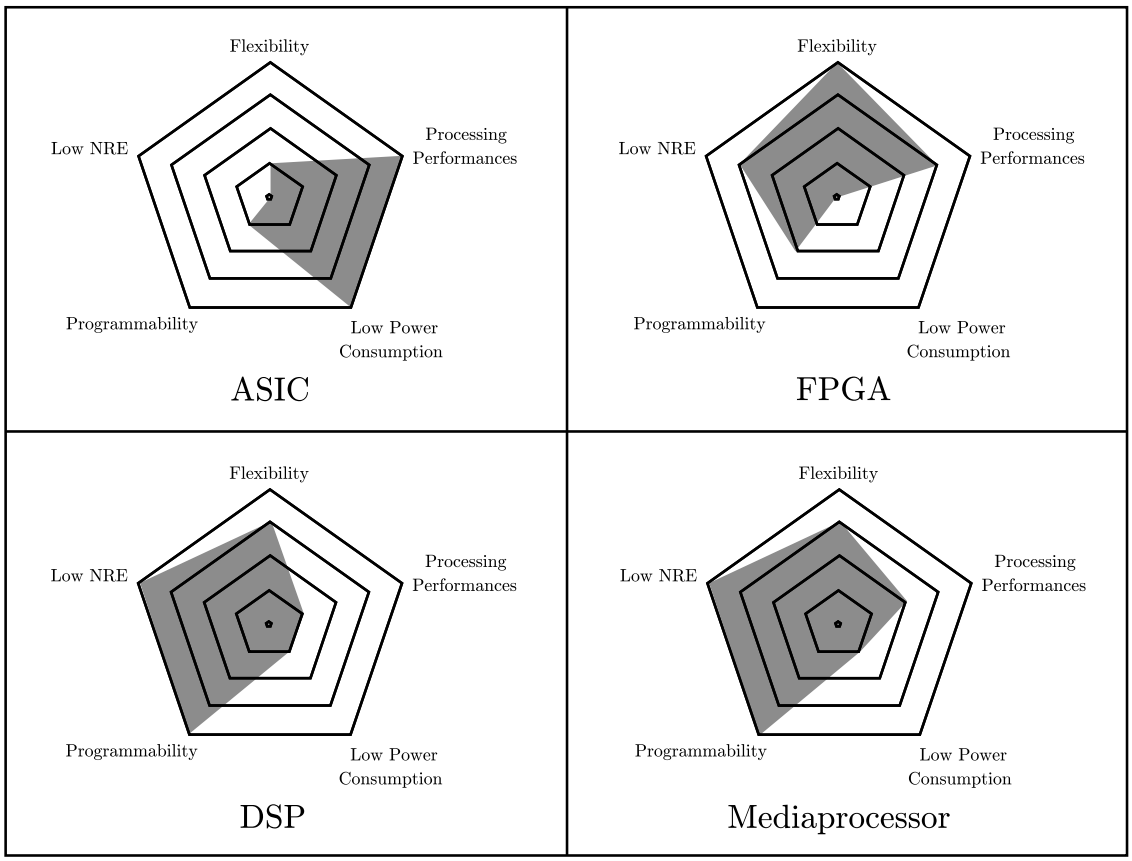


Fig. 3.6 – *Comparaison entre différents dispositifs de traitement et leur respectives caractéristiques physiques, applicatives et associées au design.*

En termes de performance de calcul et de consommation d'énergie, les dispositifs ASIC peuvent être considérés comme étant le choix idéal. Bien évidemment, le développement d'un SoC (System on Chip) dédié à une application permet d'exploiter pleinement et de façon optimale le silicium, par l'implémentation d'une architecture "custom" taillée en fonction du flot de données de l'application. Ainsi, la consommation d'énergie sera également optimisée.

Par contre, les coûts de développement (NRE) d'un tel dispositif sont très élevés, faisant de cette approche une alternative intéressante uniquement pour des volumes de production supérieurs à plusieurs milliers d'unités. En outre, en conséquence de leur spécialisation à une tâche précise, les dispositifs ASIC souffrent d'une flexibilité très basse (voire nulle), et d'une programmabilité normalement restreinte à la définition de quelques paramètres.

Les dispositifs FPGA apparaissent comme étant une excellente alternative pour les applications de haute-performance ayant des volumes de production faibles ou moyens. La technologie des circuits FPGA a connu une rapide évolution depuis plusieurs années, accompagnée d'une popularité croissante dans les milieux de l'aérospatial et militaire [64], ainsi que dans les communautés industrielle et de la recherche académique. Grâce à une augmentation du nombre d'éléments logiques (LE's) disponibles par composant, l'augmentation des fréquences d'horloge et la possibilité d'exploiter massivement le parallélisme potentiel des applications, les FPGA's sont aujourd'hui en mesure de délivrer des performances proches de celles obtenues par des circuits ASIC. En présentant bien évidemment l'avantage non-négligeable de la reconfigurabilité.

En conséquence de cette reconfigurabilité, les dispositifs FPGA apportent au système une très grande flexibilité, lui permettant de s'adapter à pratiquement n'importe quelle application. La possibilité d'implanter au sein du FPGA des éléments programmables du type "soft-core" (coeur de CPU généraliste, ou coeur de DSP), ainsi que des éléments "fixes" de type IP (Intellectual Property) viennent renforcer cette flexibilité, et constituent également un attrait important.

Par contre, la consommation d'énergie des FPGA est relativement élevé, et même si des méthodologies et environnements de développement existent, les solutions basées sur ces composants requièrent un temps de développement supérieur et un plus grand niveau d'expertise quand comparées aux solutions basées sur les dispositifs de type CPU (DSP, microcontrôleur, etc.). Les principaux fabricants de FPGA sont ALTERA (familles Cyclone et Stratix) [65] et XILINX (familles Spartan et Virtex) [66].

Les dispositifs DSP et les mediaprocessors partagent beaucoup de caractéristiques communes avec les processeurs généralistes de type RISC (PowerPC, ARM, ...) et les microcontrôleurs. Tous ces dispositifs sont en effet construits autour d'un coeur de CPU, i.e. une structure de décodage et exécution des instructions contenues au sein d'un programme. En conséquence, ces dispositifs présentent une excellente pro-

grammabilité, permettant l'utilisation de langages haut-niveau telles que C/C++, et comptant sur de nombreux outils informatiques tels que compilateurs et environnements de développement. Les coûts NRE associés à l'intégration de ces dispositifs sont faibles, et ils présentent une bonne flexibilité, pouvant être adaptés à la plupart des applications.

Les principales différences entre les différents dispositifs programmables apparaissent essentiellement au niveau des performances délivrées, et de leur spécialisation ou non à une certaine classe d'application. Les microcontrôleurs peuvent être vus comme des processeurs RISC augmentés, par l'ajout de mémoires (RAM, ROM, Flash), périphériques et interfaces d'entrée/sortie comme par exemple des ADC et DAC (convertisseur analogique-numérique et numérique-analogique, respectivement).

Les DSP en revanche présentent une architecture dédiée aux tâches de traitement du signal, ainsi que des structures matérielles optimisées pour l'exécution des opérations arithmétiques, telles les structures MAC (*multiply-accumulate*) et les unités SIMD.

Finalement, les processeurs média sont une catégorie de DSP dédiés au traitement audio/vidéo, et particulièrement adaptés au traitement des flux de données (data streaming). Les DSP et les mediaprocessors peuvent avoir des architectures de type VLIW (Very Long Instruction Word), comme par exemple le processeur TriMedia TM3270 de chez Philips [67] (figure 3.7).

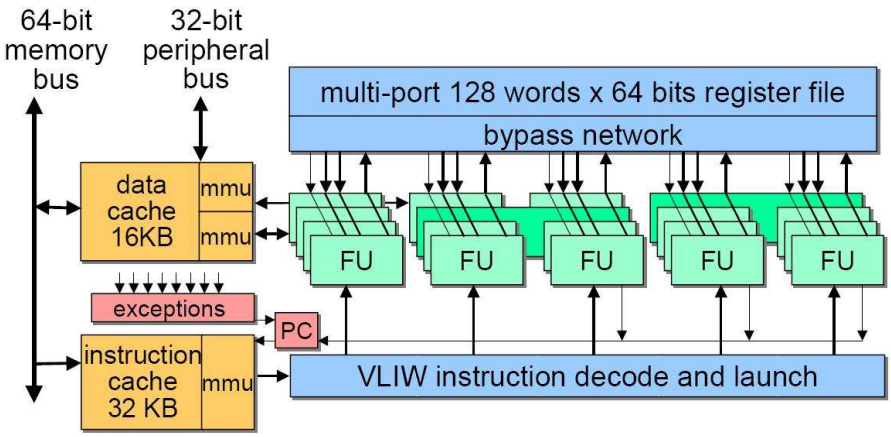


FIG. 3.7 – *Architecture VLIW/SIMD du processeur embarqué Philips Trimedia CPU-64.*

L'architecture VLIW du Trimedia comporte 5 slots, signifiant qu'en effet chaque instruction réalise jusqu'à 5 opérations simultanément. Les 128 registres de 64 bits chacun permettent l'exploitation du parallélisme SIMD, via des opérations vectorielles, et le jeu d'instructions comporte aussi des opérations dédiées au traitement des signaux, comme les MAC [68].

3.1.3 Interfaces et protocoles de communication

Plusieurs facteurs doivent être pris en considération lors du choix d'une interface de communication pour une caméra. Même si l'objectif principal d'une caméra intelligente est de traiter en interne les données acquises, afin de transmettre seulement les informations pertinentes sur la scène observée, la capacité de transmettre des images entières en pleine résolution quand ceci est nécessaire demeure un point important pour plusieurs applications. Dans ces cas, la bande passante de l'interface de communication doit être assez élevée pour supporter la transmission d'un flot vidéo, même si la caméra ne transmettra pas tout le temps à une telle cadence.

En effet, la bande passante requise est fortement liée aux caractéristiques du capteur d'images, qui supporte une certaine résolution et cadence d'acquisition de données. Il paraît être un choix judicieux que de maintenir possible l'exploitation pleine des capacités du capteur d'images. Ainsi, le système aura besoin d'une interface de communication compatible avec la cadence de sortie du capteur.

Par contre, la bande passante de transmission n'est pas la seule caractéristique importante d'un protocole de communication. Traditionnellement, une interface de caméra peut être classifiée en fonction de quatre facteurs principaux :

1. Bande passante (cadence de réception/transmission de données);
2. Compacticité, portée et câbles (filaire ou sans fil, taille des connecteurs, longueur maximale des câbles, portée sans fil);
3. Déterminisme et réactivité (latences de communication, robustesse, fiabilité);
4. Interchangeabilité (compatibilité entre fabricants, pilotes logiciels).

Les principaux protocoles utilisés sont listés dans le tableau 3.1 pour la communication filaire, et dans le tableau 3.2 pour la communication sans fil.

A titre d'exemple, si une caméra est équipée du capteur d'images MT9M413 de chez Aptina Imaging (ancienne Micron Imaging), capable de délivrer jusqu'à 660 Mpixels/s, une interface Camera Link serait nécessaire afin d'exploiter au mieux les capacités de l'imageur (5.44 Gbit/s (680 MBytes/s) dans sa *"full configuration"*).

Par contre, dans certains cas et en fonction d'autres contraintes, la règle de maintenir les cadences d'acquisition et transmission compatibles peut être abandonnée. Pour une caméra autonome fonctionnant sur piles par exemple, le protocole sans fil ZigBee peut être préférable, dû à sa très basse consommation en énergie [58, 61], même si sa bande passante de 250 Kbit/s rendra impossible la transmission vidéo en temps-réel.

Une solution possible pour réduire les exigences de bande passante sont les algorithmes de compression [60]. Néanmoins, compresser et décompresser des images représente une charge de travail supplémentaire pour la caméra et pour le système

TAB. 3.1 – *Protocoles de communication filaire.*

Protocole	Bande passante théorique (bit/s)
RS-232 serial link	19200 bit/s
USB 1.x Full-speed	12 Mbit/s
USB 2.0 Hi-speed	480 Mbit/s
FireWire or IEEE 1394a/b	400/800 Mbit/s
Camera Link	2.04, 4.08 or 5.44 Gbit/s
Ethernet, Fast Ethernet	10/100 Mbit/s
GigE Vision (Gigabit Ethernet)	1 Gbit/s

TAB. 3.2 – *Protocoles de communication sans fil.*

Protocole	Bande passante théorique (bit/s)	Portée sans fil (m)
WiFi IEEE 802.11a	54 Mbit/s	jusqu'à 10m
WiFi IEEE 802.11b	11 Mbit/s	∼50m en intérieur, ∼200m en extérieur
WiFi IEEE 802.11g	54 Mbit/s	∼27m en intérieur, ∼75m en extérieur
Bluetooth	1 Mbit/s	∼10-100m
ZigBee (IEEE 802.15.4)	250 Kbit/s	∼10-30m en intérieur, ∼150m en extérieur

hôte, et peut engendrer des pertes de qualité d'image selon le taux de compression nécessaire.

Finalement, comme mentionné auparavant, la bande passante n'est pas le seul facteur décisif. Par exemple, les coûts d'implémentation d'un système de communication Gigabit Ethernet (GigE Vision) peuvent paraître intéressants au premier abord, mais le résultat final pourrait entraver la réactivité du système (transmission des données par paquets, interruptions fréquentes, surcharge du système hôte pour le paquetage/dépaquetage), et augmenter le temps de développement. En effet, le protocole GigE Vision est très récent, pendant que des protocoles comme Camera Link et FireWire ont déjà fait leur preuves sur le terrain.

La complétude et standardisation d'un protocole doivent aussi être considérés. Le protocole USB 2.0 par exemple ne possède pas une norme standard de transmission vidéo. Les caméras GigE Vision et Firewire sont directement compatibles avec la plupart des ordinateurs récents, équipés d'usine des contrôleurs nécessaires. Ce n'est pas le cas des dispositifs Camera Link, qui requièrent l'ajout d'un *frame-grabber* dédié et coûteux.

3.2 Architectures embarquées de vision : *Smart Cameras*

L'objectif de cette section est de présenter et de décrire quelques exemples de caméras intelligentes, issues aussi bien du milieu de la recherche que du milieu industriel [51].

La caméra intelligente sans-fil WiCa (Wireless Camera) de chez NXP (ancienne Philips Research) peut intégrer un ou deux capteurs d'images couleur de résolution VGA (640 x 480, 300 kpixels). Ces capteurs sont connectés à un processeur SIMD IC3D de la famille Xetal. L'interface de communication est composée d'un module ZigBee de basse consommation, et un contrôleur ATMEL 8051 est utilisé pour la gestion de la communication et de l'opération du système [61] (figure 3.8).

FIG. 3.8 – *La caméra intelligente sans fil WiCa, de chez Philips Research Laboratories/NXP Semiconductors, Pays Bas.*

Le processeurs SIMD IC3D présente un LPA (Linear Processor Array) avec 320 éléments de traitement, alimentés par 64 mémoires ligne de 3200 bits chacune. Cet architecture s'avère extrêmement puissante pour l'exécution de routines de traitement bas-niveau, c'est à dire avec les mêmes opérations étant répétées pour chaque pixel. Par exemple, dans le cas d'une image de résolution QVGA (320 x 240), une ligne entière de l'image peut être stockée dans chaque emplacement mémoire, et traitée simultanément avec un pixel étant pris en charge par chaque élément de traitement du LPA. Outre sa puissance de calcul, un des principaux attraits de cette architecture est sa basse consommation d'énergie, qui la rend adaptée au déploiement des réseaux distribués de caméras.

Dans ce même esprit de caméra sans-fil basse consommation, on peut citer l'architecture MeshEye de l'Université de Stanford [58] (figure 3.9). Celle-ci adopte une approche originale de système multi-capteur à résolution hybride, basée sur

un imageur CMOS couleur de résolution VGA, et deux ou plus capteurs de basse-résolution (1 kilopixel), comme ceux retrouvés dans les souris optiques. Le système est complété par un microcontrôleur ATMEL AT91SAM7S, basé sur un coeur de CPU ARM7TDMI, et un module de communication sans-fil ZigBee. Ce système multi-capteurs à résolution hybride permet l'optimisation du processus d'acquisition, dans une approche typique de la vision précoce. Par exemple, un des capteurs basse résolution peut être utilisé pour la détection de mouvement dans la scène. Lorsqu'un objet mouvant est détecté, le deuxième capteur basse résolution est activé afin de réaliser un appariement stéréo, et une fois la position et la taille de l'objet mouvant estimées, une fenêtre d'intérêt contenant cet objet peut finalement être acquise par l'imageur VGA. Cette fenêtre d'intérêt pourra ensuite être traitée par la caméra elle-même, communiquée au système hôte ou échangée avec les caméras voisines dans le contexte d'un réseau distribué de caméras.

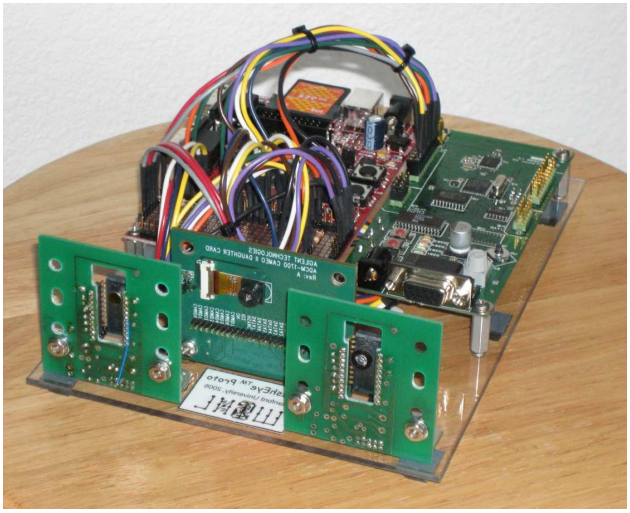


FIG. 3.9 – *Le prototype de l'architecture MeshEye, du WSNL (Wireless Sensor Networks Laboratory), à l'Université de Stanford aux États-Unis.*

Une caméra intelligente rapide a été présentée par le laboratoire Le2i de l'Université de Bourgogne [60], intégrant un capteur CMOS haute-vitesse de 1.3 Mpixels (MT9M413 de chez Aptina Imaging, ancienne Micron). Un circuit FPGA de la famille Virtex II de chez Xilinx et une interface USB 2.0 complètent le système, présenté en figure 3.10. Le capteur d'images employé est capable d'acquérir jusqu'à 500 fps (*frames per second*) en pleine résolution, impliquant dans une cadence de sortie de 6.55 Gbits/s. Afin de permettre la transmission d'un tel flot vidéo via le lien USB 2.0 (480 Mbits/s), des algorithmes de compression sont implémentés et exécutés au sein du circuit FPGA. Le taux de compression obtenu est de 30:1. D'autres tâches de traitement sont également implémentées dans l'architecture embarquée, comme le filtre de Sobel, les opérations d'érosion/dilatation et le calcul du centre de masse pour l'extraction de marqueurs.

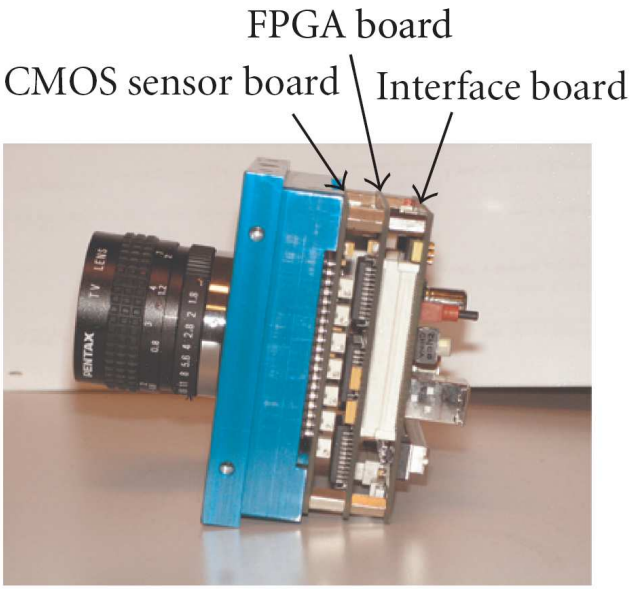


FIG. 3.10 – *La caméra rapide du laboratoire Le2i, Université de Bourgogne, France*

Un autre exemple intéressant issu du milieu académique est le projet CMUcam, porté par l'Université Carnegie Mellon [69, 57]. Ce projet a donnée naissance à trois générations successives de caméras intelligentes, toutes les trois disponibles à la vente. La "CMUcam3 Open Source Programmable Embedded Color Vision Platform" intègre un capteur CMOS couleur de résolution CIF (352 x 288), ainsi qu'un microcontrôleur NXP LPC2106, basé sur un coeur ARM7TDMI-S (figure 3.11).

Une sortie vidéo analogique est disponible, et il est également possible d'intégrer à la CMUcam un module de communication sans-fil ZigBee. La plateforme CMUcam3 inclut un environnement de développement *"open source"*, ainsi qu'une librairie de base pour la manipulation des images et d'autres outils logiciels tels qu'un interpréteur de langage, un "frame grabber" et des outils de prototypage. Il s'agit d'une architecture assez simple, mais son aspect "clef-en-main", renforcé par son environnement de développement dédié, la rend intéressante pour le déploiement d'applications simples à base de caméras intelligentes.

A l'Université de Technologie de Graz en Autriche, dans le cadre du projet SmartCam [70], une plateforme basée sur des dispositifs DSP a été développée. Cette plateforme intègre un capteur CMOS de résolution VGA et deux DSP TMS320C6415 de chez Texas Instruments [71]. L'interconnexion est réalisée via un bus PCI. Grâce à un "network processor" (Intel IXP425) embarqué, plusieurs systèmes de communication peuvent être exploités, tels que l'Ethernet, USB, RS232, WLAN et GSM. Une version évolutive pouvant contenir un réseau embarqué de jusqu'à dix composants DSP a également été proposée.

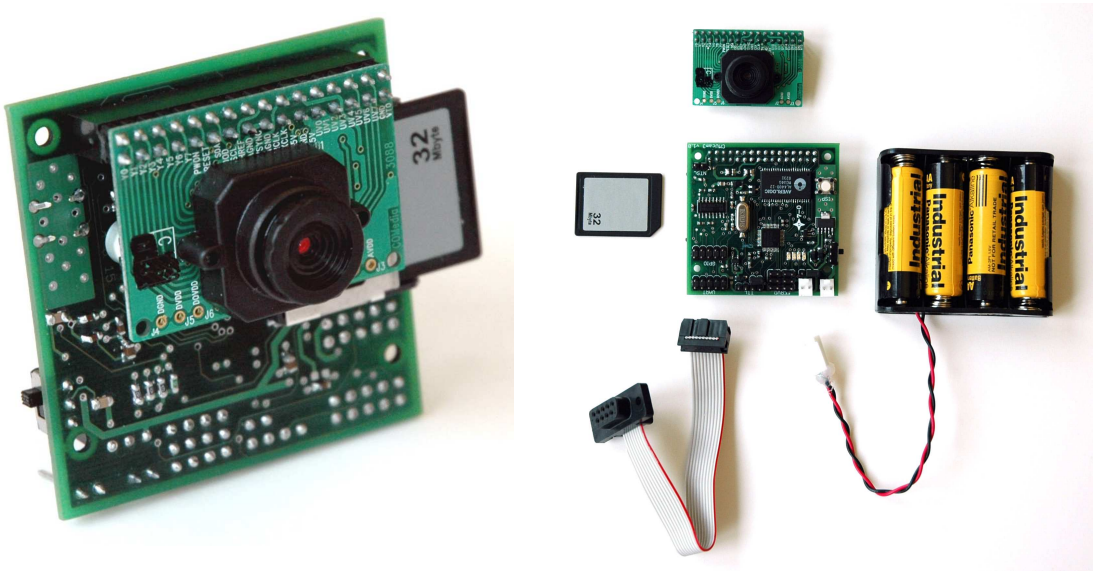


FIG. 3.11 – *La CMUcam3 de l'Université Carnegie Mellon, États-Unis.*

Parmi les applications déployées à l'aide de cette plateforme nous pouvons en citer un système de surveillance du trafic routier [59], un système de surveillance distribué [50] et un système de tracking multi-caméras [72].

FIG. 3.12 – *Prototype SmartCam de l'université de Technologie de Graz.*

Du coté industriel, les caméras intelligentes proposées dans le commerce représentent aujourd'hui des centaines de références, de la part de fabricants dans le monde entier et pour une large gamme d'applications.

L'entreprise allemande VC Vision Components propose plusieurs familles de caméras intelligentes. La famille *Optimum* par exemple (VC44xx) est composée de six modèles haute-performance, équipés de capteurs CCD de résolution allant jusqu'à 2 Mpixels, d'un DSP cadencé à 1 GHz, et d'interfaces série RS232 et Fast Ethernet, plus des entrées/sorties numériques pour le contrôle directe d'équipements externes.

D'autres exemples sont les caméras intelligentes INCA et son successeur DICA du fabricant hollandais Philips (fig. 3.13). Ces caméras sont munies d'un processeur VLIW Trimedia, d'une mémoire flash pour stocker les applications et de plusieurs interfaces de connexion. Elles sont capables de réaliser en interne le traitement des images acquises, et peuvent fonctionner de façon autonome, sans être branchés à un PC. L'architecture de ces caméras est montrée dans la figure 3.14. Le capteur d'images pour le modèle DICA 321 est un CMOS monochrome de résolution SXGA (1280 x 1024). Le CPLD sert à commander le capteur, qui peut fonctionner à des fréquences allant jusqu'à 40 MHz, avec un frame-rate de 30 images/s à pleine résolution.

FIG. 3.13 – *Les caméras intelligentes INCA et DICA de Philips.*

Dans les travaux présentés dans [73], deux caméras DICA 321 sont utilisées pour constituer une plateforme de stéréo-vision. L'application proposée est le calcul de disparité entre deux images, afin d'obtenir une estimation de la profondeur de la scène, cependant la plateforme est conçue pour être un système généraliste de vision et traitement d'images. Les caméras sont légèrement modifiées, avec l'ajout d'un port bidirectionnel au niveau du CPLD, permettant l'interfaçage direct des pixels avec un circuit FPGA. Un processeur SIMD Xetal est également intégré à l'architecture (fig. 3.15).

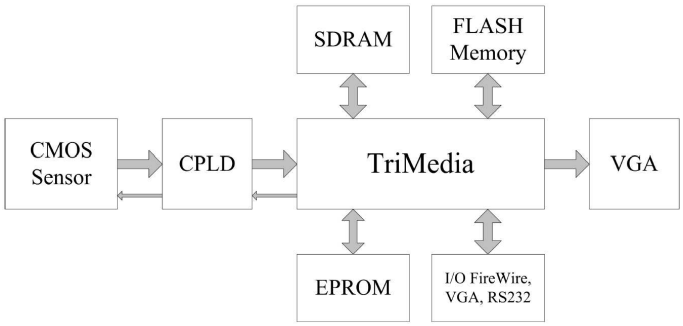


FIG. 3.14 – *Architecture interne des caméras INCA et DICA.*

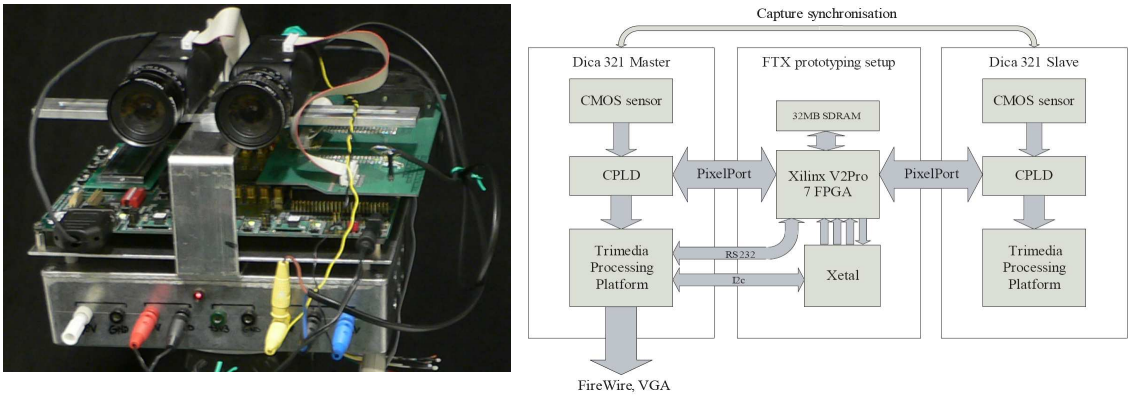


FIG. 3.15 – *Plateforme de stéréo-vision temps-réel et son architecture.*

La division Smart Systems, du Austrian Research Centers GmbH, propose un capteur intelligent de surveillance du trafic [74, 75] et un capteur intelligent de comptage de personnes. Ces deux dispositifs sont basés sur une nouvelle technique d'imagerie constituée de pixels autonomes qui répondent aux changements relatifs de l'intensité de lumière. Cette technologie est spécialement adaptée à la détection d'objets mouvants, étant robuste à l'illumination de la scène.

L'entreprise américaine National Instruments distribue cinq modèles de caméras intelligentes (NI 17xx), avec des capteurs CCD monochrome (résolution VGA ou SXGA), un processeur embarqué PowerPC et une interface Gigabit Ethernet. Les modèles 1762 et 1764 intègrent également un DSP Texas Instruments cadencé à 720MHz en tant que co-processeur. Des contrôleurs d'illumination intégrés permettent le contrôle de l'éclairage des objets sous la caméra. Cette caractéristique peut être particulièrement utile pour des applications de vision industrielle telles que le contrôle de qualité. Un environnement de développement est disponible (NI Vision Builder for Automated Inspection), ainsi qu'un kit de développement pour la programmation de la caméra avec LabView.

L'entreprise suédoise SICK IVP a deux modèles de smart caméras pour des

environnements industriels : la IVC-2D et la IVC-3D. Les deux sont équipées d'un processeur cadencé à 800MHz, d'un FPGA comme accélérateur matériel, et d'une interface Fast Ethernet. La IVC-2D présente un capteur CCD de résolution VGA ou XGA (1024 x 768).

La IVC-3D présente un capteur CMOS optimisé pour l'imagerie 3D. Cette caméra est capable de réaliser des mesures d'épaisseur au moyen d'un système laser intégré dans le boîtier de la caméra, et d'une technique de triangulation. Le laser dessine une ligne sur l'objet, et la caméra, qui regarde cette ligne d'un certain angle, verra une courbe qui suivra le profil de hauteur de cet objet. Quand ceci passe sous le faisceau, une image tridimensionnelle est construite à partir de plusieurs profils de hauteur. La programmation et configuration de la caméra sont faites au moyen de l'outil de programmation *user friendly* IVC-Studio.

D'autres exemples sont les caméras intelligentes Sony XCI et Intellio ILC, illustrées en figure 3.16. La caméra Sony XCI-SX1 intègre un capteur CCD de résolution SXGA et un processeur AMD Geode GX533 cadencé à 400MHz. Le processeur héberge un système d'exploitation temps-réel Linux MontaVista, afin d'assurer la performance et la flexibilité de la plateforme. Plusieurs interfaces de communication sont disponibles, parmi elles une sortie analogique VGA, ainsi que des liens Ethernet, USB et RS232. Cette caméra permet de développer des solutions pour une grande variété d'applications de vision industrielle.

FIG. 3.16 – *A gauche la caméra intelligente Sony XCI-SX1, pour des applications de vision industrielle, et à droite la caméra intelligente Intellio ILC-210, pour les systèmes de sécurité et surveillance.*

La caméra ILC-210B/E du fabricant hongrois Intellio est dédiée aux applications de sécurité et surveillance. Avec un capteur CMOS de résolution XGA, elle est capable d'opérer en conditions de jour et de nuit. Plusieurs tâches de détection d'événements peuvent être réalisées, telles que la détection de mouvement, détection d'objets abandonnés, détection d'intrusion, analyse de foule, etc. Le même fabricant propose également des caméras pour la surveillance du trafic.

Nous citerons pour clore cette section un exemple particulier de caméra intelligente. La "NeuriCam VISoc CMOS Intelligent Vision System-on-Chip" est en effet une caméra intelligente entièrement intégrée [76]. Dans un même chip sont intégrés un capteur CMOS de résolution 320 x 256, un processeurs RISC, un co-processeur de vision, ainsi que des dispositifs d'entrée/sortie. Ce haut degré d'intégration présente des avantages majeurs pour des applications avec des contraintes importantes de taille (UAVs par exemple) ou de consommation d'énergie.

Les inconvénients sont un manque de flexibilité, un design moins modulaire [77], et un coût de fabrication plus élevé quand comparé aux coûts de fabrication des capteurs d'images standard.

Comme nous pouvons constater, les caméras intelligentes peuvent présenter des architectures très variées : monoprocesseur, multi-DSP, mixte processeur-DSP, hétérogène, etc. La palette des dispositifs employés est également très large : processeurs embarqués, DSP's, FPGA's, mediaprocessors, processeurs SIMD, etc.

Le champ d'application comprend, entre autres, le contrôle du trafic, les réseaux distribués, la vidéo-surveillance, la mesure 3D, le contrôle de qualité, la vision rapide et la vision stéréo. Ce sont les contraintes imposées par le champ d'application envisagé qui permettront de guider le choix de l'architecture la plus adaptée.

Dans la suite nous présentons en détail la caméra intelligente SeeMOS, dédiée aux applications de vision active et précoce, et qui a été utilisée comme support expérimental pour cette thèse.

3.3 La plateforme SeeMOS

Le projet SeeMOS est un projet de recherche mené au LASMEA. Son principal objectif est de proposer une plateforme de recherche dédiée à la vision active, et particulièrement aux tâches de vision précoce. Ce projet a consisté dans un premier temps dans le développement d'un prototype de caméra intelligente basée sur un imageur CMOS et un dispositif FPGA [31]. Ce prototype est illustré en figure 3.17.

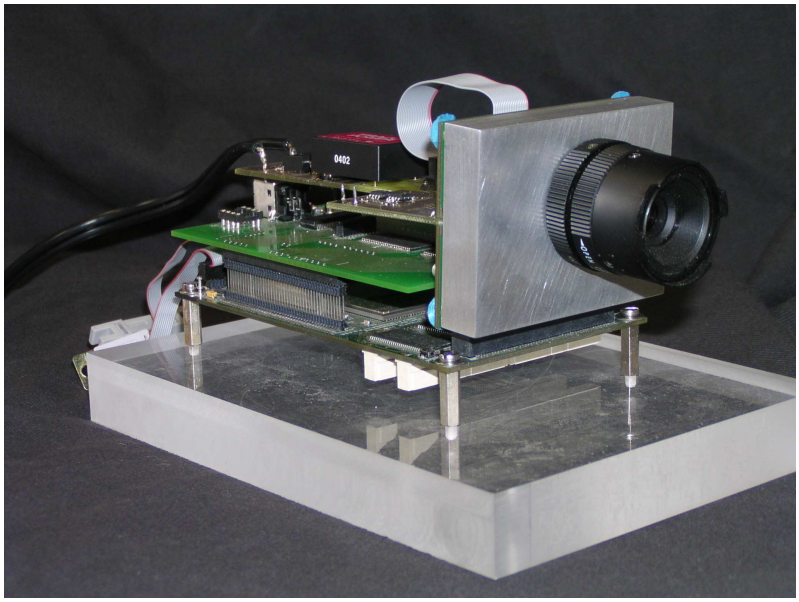


FIG. 3.17 – *Prototype de la caméra intelligente SeeMOS.*

La suite du projet, dans laquelle s'incluent les travaux réalisés dans le cadre de cette thèse, est consacrée au développement d'une méthodologie de design et d'implémentation de systèmes de vision précoce dans les architectures embarquées. Les cibles matérielles d'implémentation visées sont les dispositifs du type caméra intelligente basés sur un élément de traitement reconfigurable (FPGA). Pour cela, la caméra intelligente SeeMOS a été utilisée en tant que support expérimental de ces travaux.

La caméra SeeMOS est une architecture matérielle modulaire de traitement embarqué. La version actuelle de la caméra (figs. 3.18 et 3.19) est composée de :

- un capteur d'images CMOS ;
- une unité inertielle ;
- un dispositif FPGA ;
- 5 blocs externes indépendants de mémoire SRAM ;
- une carte de *co-processing* munie d'un dispositif DSP ;
- une interface de communication Firewire.

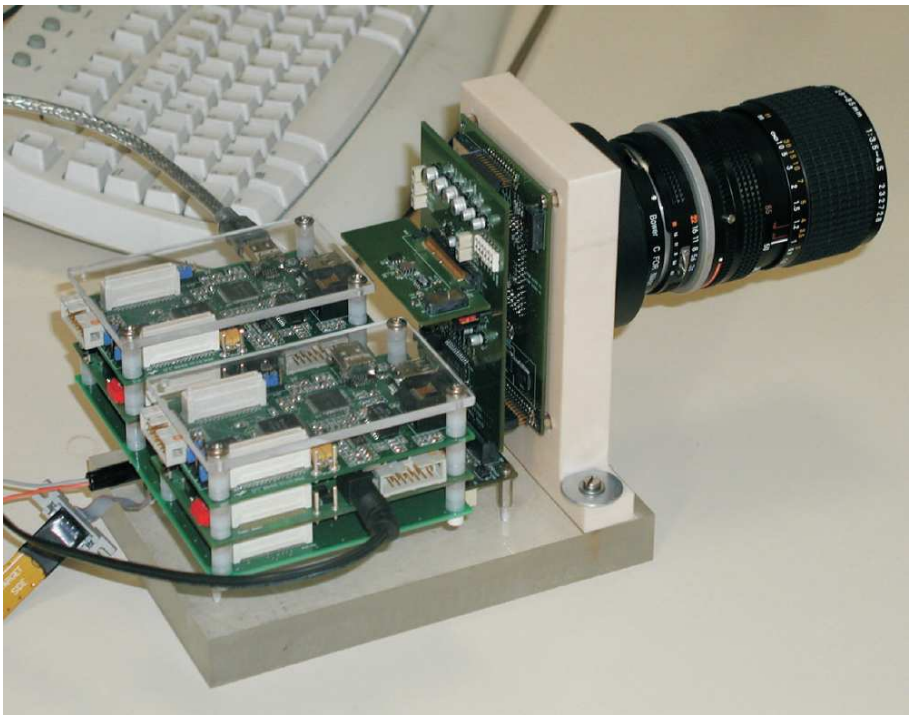


FIG. 3.18 – *La caméra intelligente SeeMOS.*

Le capteur CMOS est un modèle LUPA-4000 de chez Cypress Semiconductor. Il s'agit d'un capteur d'images monochrome et de résolution 4 Mpixels (2048 x 2048). Il est capable d'acquérir jusqu'à 66 Mpixels par seconde, chaque pixel codé sur 10 bits. L' acquisition est faite en mode "global shutter" (ou *obturateur global*), évitant les problèmes associés au mode "rolling shutter" de certains imageurs CMOS. La fréquence d'acquisition permet l'obtention d'un *frame rate* de plus de 200 fps pour une résolution VGA (640 x 480).

Le choix d'un imageur CMOS se justifie principalement par ses capacités d'adressage aléatoire. Ceci s'avère extrêmement utile pour des applications telles que le suivi d'objets, avec la possibilité d'acquérir uniquement une fenêtre d'intérêt contenant l'objet à suivre. L'adressage aléatoire permet donc d'allier, au sein d'un même dispositif, la haute-résolution du capteur à une haute cadence d'acquisition si nécessaire, par l'adressage uniquement d'une petite portion de la matrice photosensible. La conception de la caméra SeeMOS permet d'exploiter pleinement l'adressage aléatoire, obtenant par exemple des cadences de 1000 fps pour des images de résolution 140 x 140 pixels.

L'unité inertielle est composée de 3 accéléromètres, alignés sur les axes X, Y, et Z du capteur, et de 3 gyroscopes. Les données inertielles permettent l'estimation des mouvements 3D de la caméra (*ego-motion*), ainsi que de son orientation et de sa position.

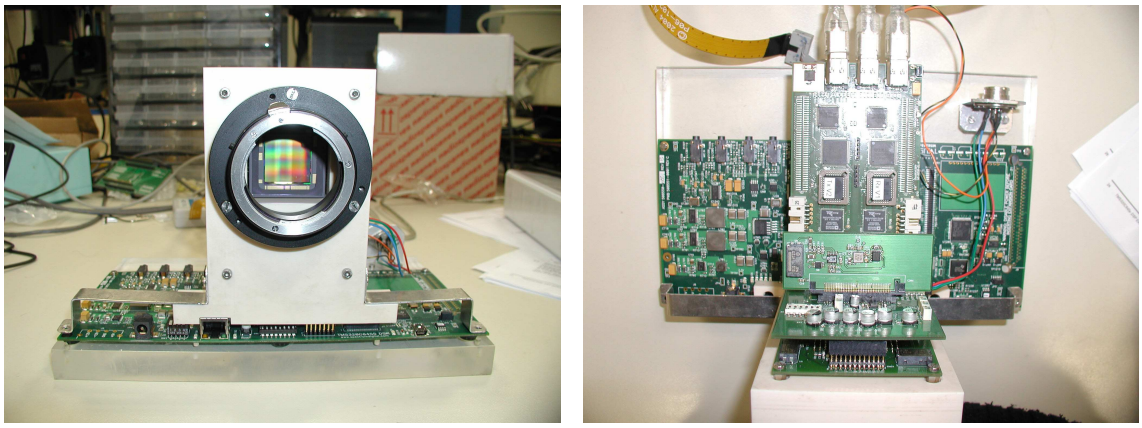


FIG. 3.19 – *La caméra SeeMOS, développée au LASMEA.*

L'ensemble des composants de la plateforme sont connectés au dispositif FPGA (fig. 3.20). Ce dernier, de la famille ALTERA Stratix modèle EP1S60F1020C7, joue un rôle central dans le système, étant responsable de l'interconnexion, du contrôle et de la synchronisation des modules sensoriels (imageur + unité inertielle), des RAMs externes, ainsi que des cartes de communication (interface Firewire) et de *co-processing* (carte DSP). Les principales caractéristiques du FPGA utilisé sont détaillées dans le tableau 3.3.

Modèle	**Altera Stratix EP1S60F1020C7**
LEs (Logic Elements)	57.120
M512 RAM blocks (32 x 18 bits)	574
M4K RAM blocks (128 x 36 bits)	292
M-RAM blocks (4K x 144 bits)	6
Total RAM bits	5.215.104
DSP blocks	18
Embedded multipliers (9 x 9-bit)	144
PLLs (Phase-Locked Loops)	12
Package	1.020-Pin FineLine BGA
User I/O pins	773
Pitch (mm)	1,00
Area (mm^2)	1.089
Length x width (mm x mm)	33 x 33
Speed grade	-7

TAB. 3.3 – *Caractéristiques du dispositif FPGA intégré dans la caméra intelligente SeeMOS.*

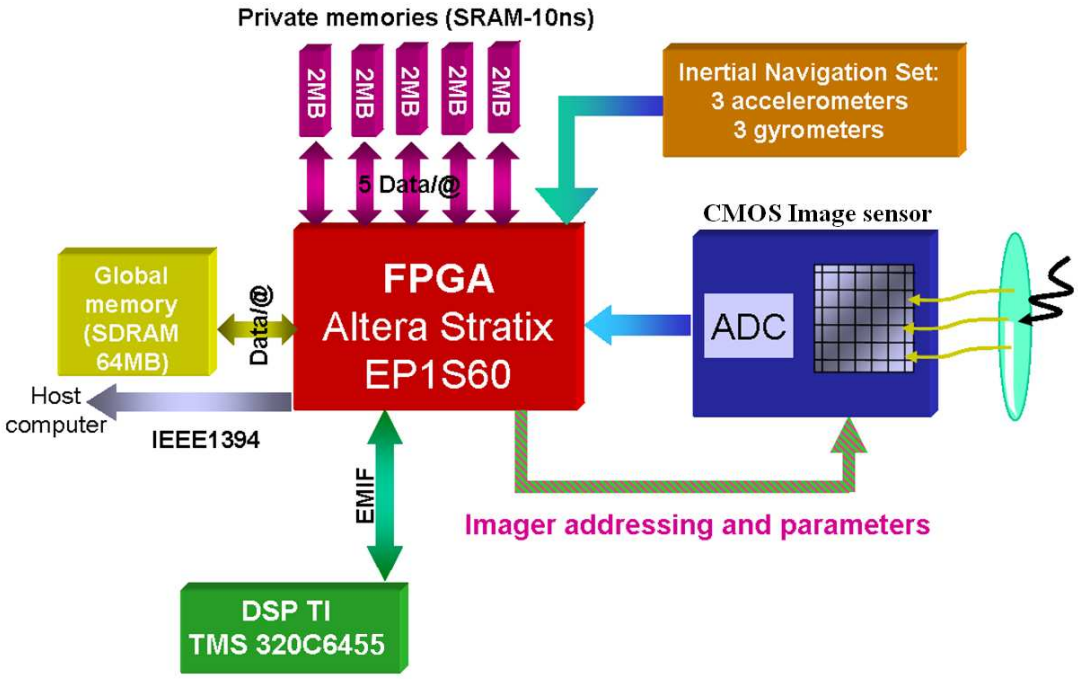


FIG. 3.20 – *Schéma synoptique de l'architecture matérielle de la plateforme SeeMOS.*

La carte FPGA (fig. 3.21) intègre également 5 blocs de mémoire RAM statique. Chaque bloc a une capacité de stockage de 2 Mo, divisés en 1M mots de 16 bits. Chacun d'entre eux dispose de son propre bus d'adresse et données. Ceci permet des accès concurrents aux différents blocs, constituant une caractéristique essentielle afin d'exploiter pleinement les possibilités de parallélisme offertes par le circuit FPGA. Un connecteur situé sur le dessous de la carte permet l'ajout d'un bloc supplémentaire de mémoire SDRAM (SODIMM 144 broches, 133/100 MHz), d'une capacité de jusqu'à 64 Mo.

L'utilisation d'une unité de *co-processing* basée sur un DSP est justifiée par le besoin pour certaines d'applications d'exécuter des routines mathématiques complexes. Celles-ci peuvent s'avérer particulièrement difficiles à implémenter en logique câblée (*hard-wired*), au moyen d'un langage HDL. D'autre part, la limitation des fréquences d'horloge dans les circuits FPGA pourrait rendre leur exécution inefficace au sein d'un élément programmable de type *soft-core*.

Une carte DSK (DSP Starter Kit) de chez Texas Instruments a donc été intégrée à la plateforme. Cette carte dispose d'un dispositif DSP modèle TMS320C6455-1000, basé sur l'architecture VLIW VelociTI. Il s'agit d'un DSP virgule fixe, fonctionnant à une fréquence d'horloge de 1GHz. Il possède un cache L1 de 64Ko (32Ko programme, 32Ko données), et un cache L2 de 2Mo.

Grâce à son architecture VLIW, basée sur huit unités fonctionnelles, le coeur

FIG. 3.21 – *Photo de la carte contenant le composant FPGA ALTERA Stratix EP1S60. A gauche nous pouvons voir le connecteur de la carte sensorielle, et quatre des cinq blocs SRAM disponibles.*

du DSP est capable d'exécuter jusqu'à 8 instructions de 32 bits par cycle, résultant dans une cadence maximale de 8000 MIPS. Deux des huit unités fonctionnelles sont dédiées aux opérations de multiplication et multiplication-accumulation (MAC). Chacune est capable de réaliser jusqu'à 4 opérations MAC par cycle (16 bits x 16 bits). Il est donc possible d'obtenir des pics de traitement de jusqu'à 8000 MMACs (Millions de MACs) par seconde. La communication entre le DSP et le FPGA se fait au moyen du protocole EMIF (Extended Memory Interface).

L'interface entre la caméra et le système hôte est assurée par un module Firewire (IEEE 1394), délivrant un débit descendant (caméra vers PC) de 20 Mo/s, et un débit ascendant (PC vers caméra) de 10 Mo/s. Ces valeurs sont des valeurs efficaces, i.e. pouvant être effectivement exploités pour le transfert de données entre la caméra et le système hôte.

Une de forces majeures de la plateforme matérielle SeeMOS provient en effet de sa modularité, les différents éléments matériels étant intégrés dans différentes cartes (fig. 3.22). Ceci facilite fortement l'évolution de la plateforme, les cartes pouvant être aisément remplacées. On peut donc procéder à une modification ou *upgrade* d'une partie du système, sans pour autant le re-concevoir dans son intégralité. Grâce à cette caractéristique la plateforme a connu des évolutions constantes, comme en témoignent les figures 3.17, 3.18 et 3.19. D'autres évolutions sont actuellement en vue, notamment l'intégration d'une carte de *co-processing* DSP dédiée (à la place du DSK Texas), et le passage vers une nouvelle génération de composant FPGA (Stratix III ou Stratix IV).

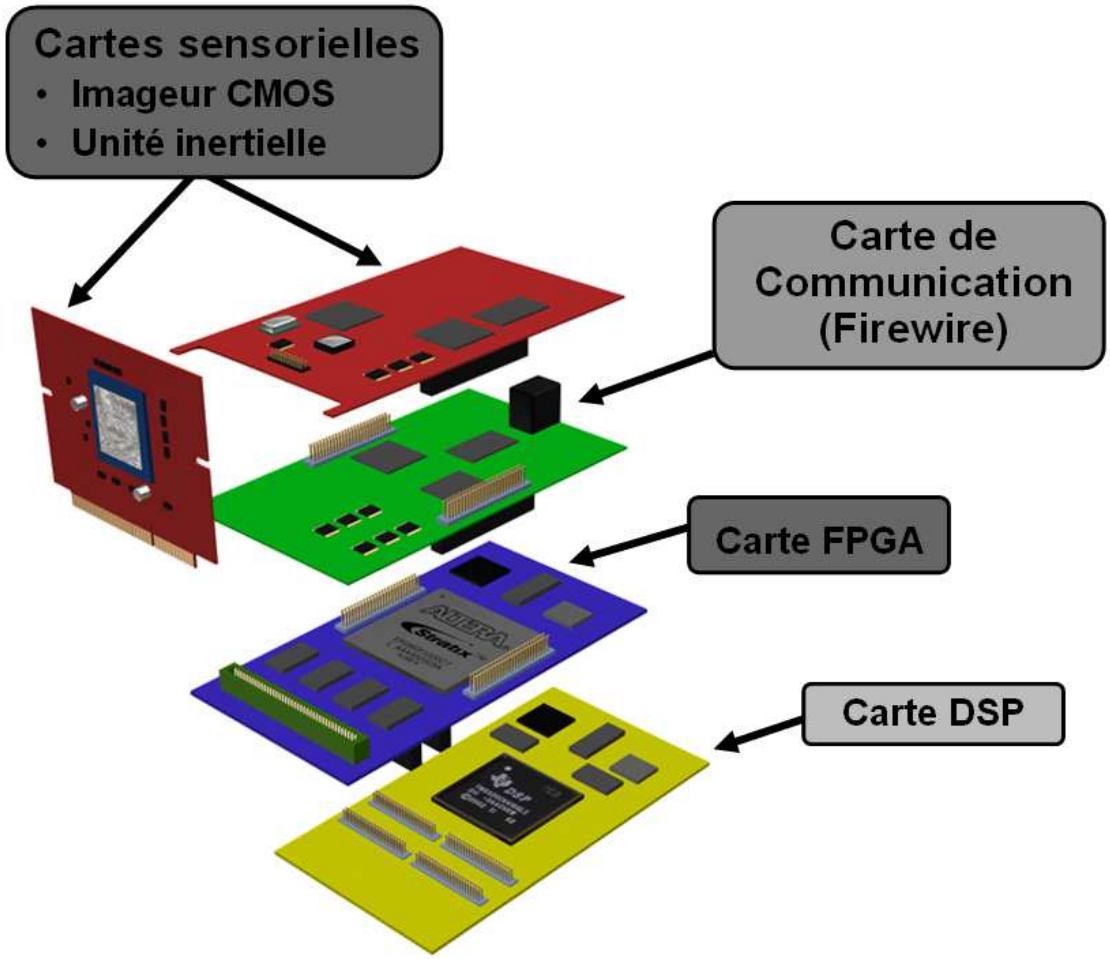


FIG. 3.22 – *Illustration des différentes cartes composant la plateforme hétérogène SeeMOS, ainsi que de sa conception modulaire.*

Les différentes caractéristiques présentées ci-dessus font de la caméra intelligente SeeMOS une plateforme puissante d'acquisition et traitement d'images. Par contre, le nombre important d'éléments matériels et leur hétérogénéité soulèvent un certain nombre de problèmes liés à la gestion des ressources matérielles. La programmation d'une telle plateforme requiert un degré élevé de connaissance du système, ainsi que la maîtrise de différents langages de programmation et environnements de développement. En conséquence, l'implémentation d'une application donnée au sein du système peut constituer un défi considérable.

La suite de cette thèse est consacrée aux outils et méthodologies permettant de relever ce défi, d'abord par la présentation et l'analyse d'un certain nombre de méthodologies existantes, et puis par la proposition d'une méthodologie originale développée dans le cadre de mes travaux de recherche.

Chapitre 4

État de l'art : Méthodologies de développement

"There is always an easy solution to every human problem - neat, plausible, and wrong."

"Il y a toujours une solution facile à tous les problèmes de l'humanité - simple, plausible, et fausse."

Henry Louis Mencken, journaliste et essayiste américain (1880-1956).
The Divine Afflatus (1917).

Les évolutions et perfectionnements matériels et architecturaux ont abouti aujourd'hui à un vaste catalogue de dispositifs dédiés au traitement de l'information. Font partie de ce catalogue les circuits FPGA, processeurs DSP, processeurs embarqués généralistes, accélérateurs matériels et IP's en tout genre. On a aussi assisté à l'apparition de protocoles de communication rapides (USB, FireWire, CamLink), et au développement et à la démocratisation des technologies d'imagerie numérique (capteurs CCD et CMOS). Tous ces facteurs réunis ont rendu possible la conception de systèmes embarqués de plus en plus performants, mais aussi de plus en plus complexes. Cette complexité accrue est une conséquence de plusieurs facteurs, parmi lesquels nous pouvons citer :

- la présence de plusieurs éléments de calcul au sein du système ;
- la possible hétérogénéité de ces éléments ;
- l'interconnexion et la synchronisation des différentes parties du système ;
- l'hétérogénéité au niveau des flots de conception pour différents éléments de calcul ;
- des langages de programmation spécifiques à chaque élément.

Afin de pallier cette complexité, une méthode et des outils adaptés au processus de design et d'implémentation sont nécessaires. Le but principal d'une telle méthode est de rendre les aspects matériels du système les plus transparents possibles au programmeur d'applications. Ainsi, la méthodologie de design, accompagnée des outils adaptés, devra permettre l'abstraction de la complexité et de la possible hétérogénéité du système.

Ainsi, une bonne méthodologie de conception doit permettre de faire le pont entre le *software* et le *hardware*, afin que le système puisse être programmé avec peu ou pas de connaissance sur le matériel. Ceci est fortement conseillé, voire nécessaire, puisque le programmeur d'applications n'est pas forcément un expert en électronique numérique embarquée.

En d'autres termes, elle doit permettre une "prise de distance" par rapport aux spécificités matérielles. Ce recul permet de généraliser en quelque sorte la description de l'application, la rendant la plus indépendante possible de la plateforme matérielle cible. Ainsi, une même description (ou programme) pourrait servir à implémenter une application donnée sur différentes plateformes. Ce chapitre est consacré à l'étude de telles méthodologies par la présentation et l'analyse des méthodologies existantes.

Comme il a été présenté dans le chapitre précédent, il existe un grand nombre de modèles de caméras intelligentes disponibles dans le commerce. Parallèlement, il existe également un certain nombre de projets "non-commerciaux" et de prototypes "faits maison", le plus souvent issus du milieu de la recherche académique, telle la plateforme SeeMOS développée au LASMEA.

Les développeurs en vision embarquée peuvent donc aujourd'hui acquérir une

"Smart Camera" auprès d'un des nombreux fabricants, afin d'y implémenter leurs propres applications taillées sur mesure. Dans ce cas, la programmation de la plateforme est faite le plus souvent par l'utilisation d'un environnement de développement intégré (IDE - Integrated Development Environment), fourni (ou vendu) par le fabricant de la caméra. Ces IDE's sont dédiés à une famille ou modèle spécifique de *smart camera*, et sont souvent basés sur un modèle de programmation graphique, permettant le développement simplifié et rapide des applications. Néanmoins, cette simplification du processus de développement engendre inévitablement une perte de flexibilité, limitant le domaine d'application aux fonctionnalités prévues par le fabricant du dispositif.

D'un autre coté, dans le cas des plateformes de recherche ou "non-commerciales", souvent conçues *ex nihilo*, l'absence d'outils de développement dédiés rend le processus d'implémentation long et complexe, car celui-ci exige :

- d'une part, la description de l'application par la programmation/configuration directe et individuelle du ou des éléments de traitement (e.g. microprocesseur, DSP, FPGA);

- d'autre part, la gestion bas-niveau des aspects électroniques de la plateforme (contrôle des capteurs et ADC's, etc.).

Il existe heureusement un certain nombre de méthodes et outils s'adressant aux problèmes posés ci-dessus, et notamment à la programmation/configuration des éléments de traitement. La suite de ce chapitre est consacrée à la présentation et à l'analyse de certaines de ces méthodes, mais d'abord nous aborderons brièvement le sujet des environnements intégrés de développement.

4.1 Les environnements de développement dédiés

Comparés aux méthodes "classiques" de programmation (écriture d'un programme pour un dispositif de traitement de données), les environnements de développement dédiés proposent une approche du processus d'implémentation plus intuitive, mais aussi moins flexible. Il s'agit d'environnements disposant d'une GUI (*Graphic User Interface*), et le plus souvent basés sur l'interconnexion de fonctions pré-existantes sous la forme de blocs (description structurelle, à l'instar de l'extension *Simulink* du logiciel MatLab). La conception du programme revient donc à la création d'un schéma graphique mettant en relation les différentes étapes de traitement, représentées par des blocs indépendants, et leur respectifs ports d'entrée et sortie. La programmation peut donc être faite sans connaissance particulière des aspects matériels de la caméra, ni sur les subtilités mathématiques inhérentes à chaque

tâche de traitement élémentaire. Par contre, le développement des applications est limité par la palette de fonctionnalités pré-existantes proposées dans la bibliothèque de l'environnement.

L'avantage majeur d'une telle approche par rapport à la programmation directe réside dans un processus de développement plus court et plus facile, accessible également à des non-programmeurs. La facilité de programmation est en effet inversement proportionnelle à la flexibilité de l'outil. Ainsi, un outil proposant des fonctionnalités (blocs élémentaires) de très haut-niveau permettra de programmer une application avec très peu d'effort, mais souffrira d'une flexibilité restreinte.

Un exemple d'IDE est l'environnement IVC Studio [78], proposé par l'entreprise suédoise SICK IVP pour les modèles de Smart camera IVC-2D et IVC-3D (illustré en figure 4.1). Il s'agit d'un environnement de programmation graphique où les différents outils de traitement d'images sont sélectionnés à l'aide d'icônes, et peuvent être paramétrés facilement par le remplissage de champs prévus à cet effet. L'environnement propose une bibliothèque d'environ une centaine d'outils de traitement d'images, concernant l'acquisition, l'affichage, la sélection de zones d'intérêt, la détection de bords, le filtrage, le calcul et etc.

L'outil peut être utilisé pour créer un programme, le tester en mode débogage, l'implémenter dans la caméra pour un fonctionnement en mode autonome, et finalement pour recueillir des informations concernant les résultats et statistiques des traitements et analyses implémentées. Ce genre d'outil permet un temps de développement court, par le prototypage et débogage rapide des applications.

Un deuxième exemple, provenant du milieu de la recherche cette fois ci, est l'environnement CMUcam2 GUI, proposé gratuitement sur Internet pour les utilisateurs de la caméra intelligente CMUcam2, développée par l'Institut de Robotique de l'Université Carnegie Mellon aux États-Unis.

Le CMUcam2 GUI (figure 4.2 [79]) est une application Java conçue pour permettre l'exploitation et le contrôle des différentes fonctionnalités embarquées dans la caméra CMUcam2. Parmi ces fonctionnalités nous pouvons en citer le suivi de zones de couleur, la détection du mouvement et le calcul d'histogrammes. Ces différents traitement sont en effet pré-implémentés dans la plateforme matérielle, et l'interface graphique permet de les configurer afin de prototyper rapidement et facilement des applications de vision.

L'intérêt de ce genre d'approche est notamment, comme il a déjà été cité, de permettre aux non-programmeurs de concevoir des applications de vision de façon simple et rapide. Également, une connaissance approfondie du matériel n'est pas nécessaire. Par contre, la limitation du développement aux bibliothèques et fonctions pré-existantes représente un inconvénient majeur, restreignant fortement le champ d'application.

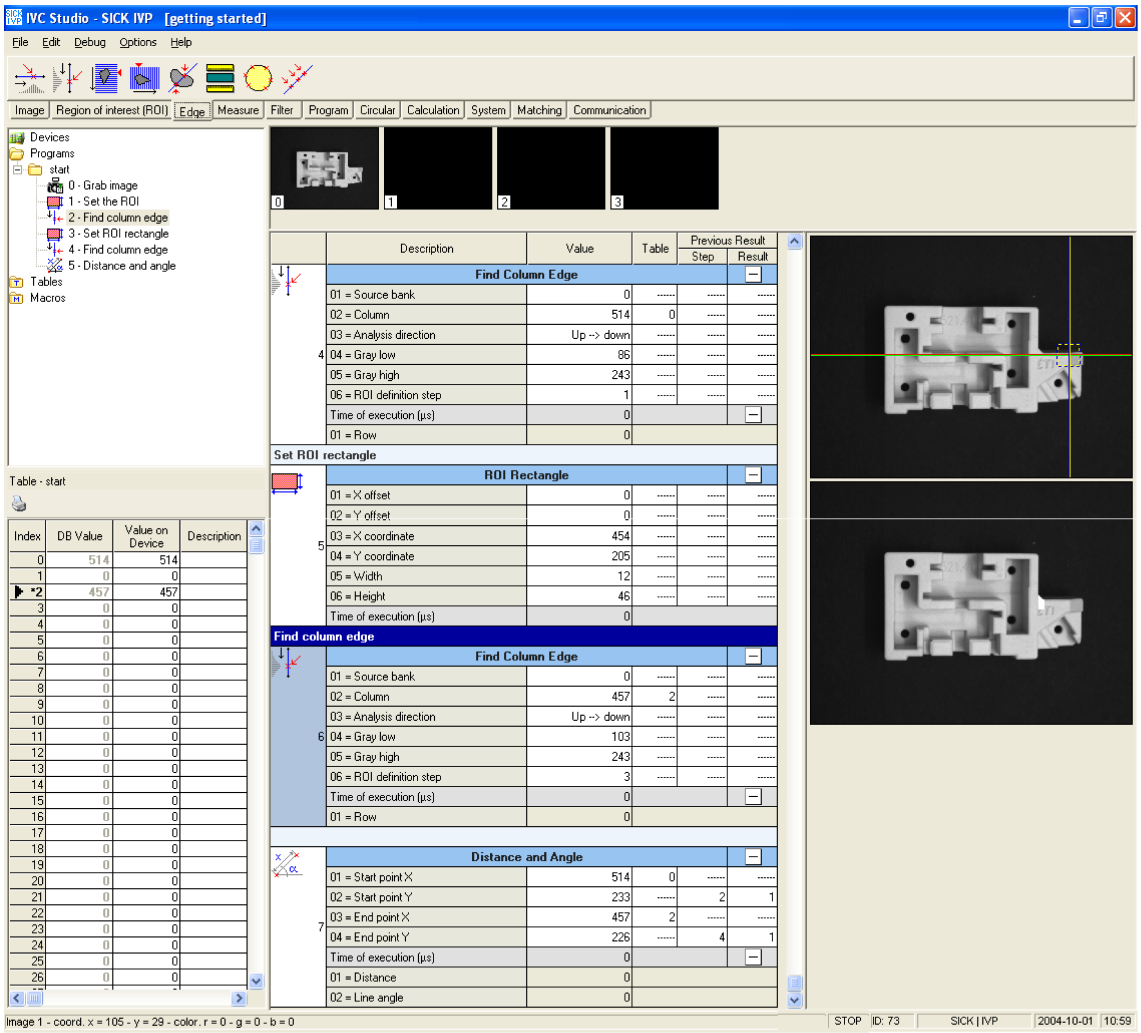


FIG. 4.1 – *Capture d'écran de l'environnement de développement IVC-Studio, proposé par Sick IVP pour les caméras IVC-2D et IVC-3D.*

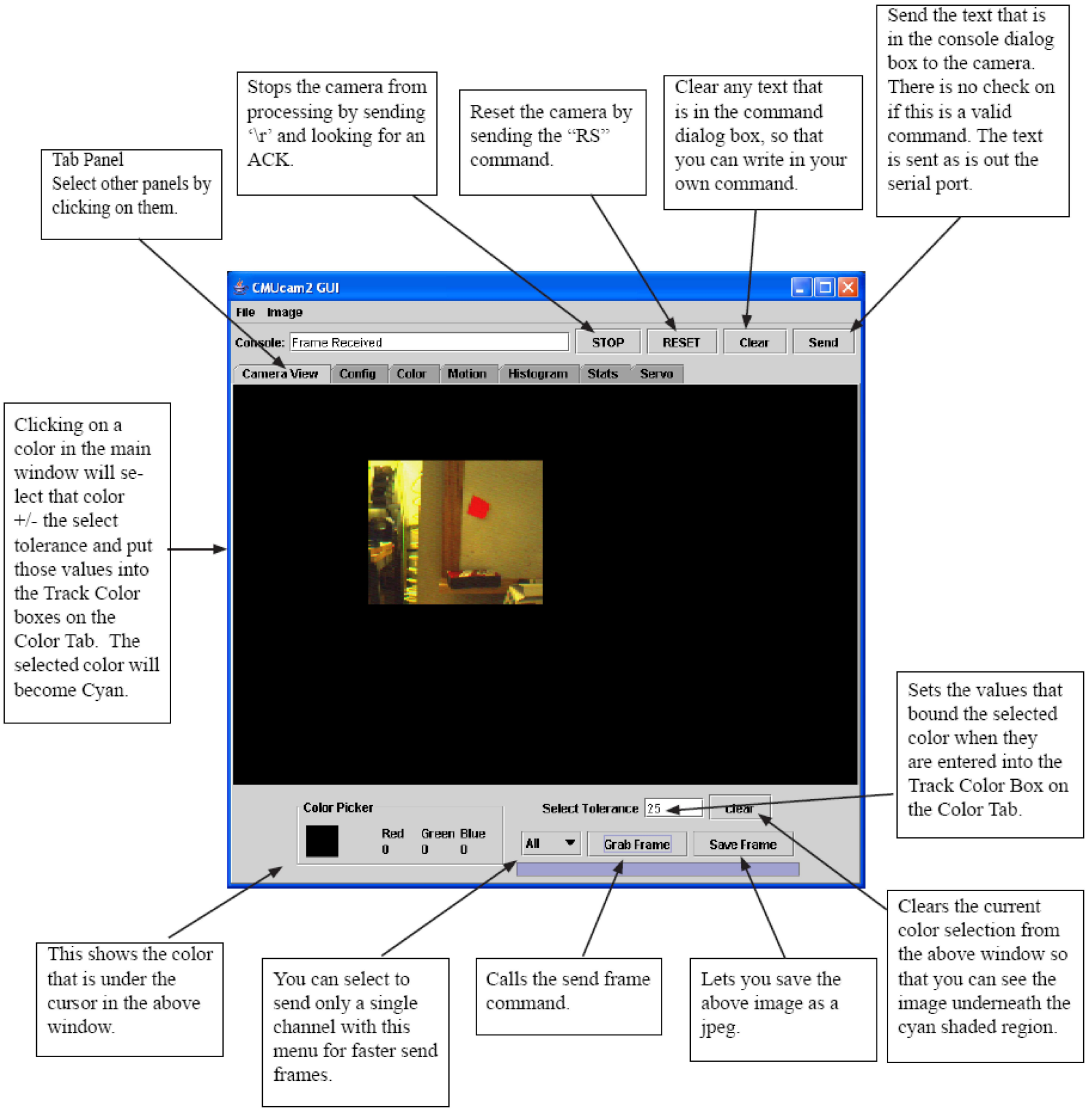


FIG. 4.2 – *Capture d'écran de l'applicatif d'acquisition, programmation et débogage proposé avec la caméra CMUcam2.*

4.2 Méthodologies et outils de développement et programmation

En l'absence d'environnement de développement dédié, et en dépendant des caractéristiques matérielles de la caméra intelligente en question, le développeur d'applications sera confronté à une ou plusieurs des tâches suivantes :

- écrire ou générer du code pour un ou plusieurs éléments programmables (microprocesseur, microcontrôleur, DSP, etc.) ;
- écrire ou générer du code HDL (Hardware Description Language) pour les éléments reconfigurables (FPGA) ;
- exprimer les dépendances de données entre les fonctions implantées dans différents éléments, et éventuellement exploiter les parallélismes ;
- réaliser le "mapping" entre les fonctions et les éléments, quand plusieurs possibilités d'implantation existent ;
- gérer les échanges (données et contrôle) entre les différents éléments de la plateforme (synchronisation).

Les deux premiers points concernent simplement la programmation des éléments de traitement. Dans le cas des éléments programmables (basés sur un coeur de processeur), les langages le plus souvent employés sont le C ou le C++, qui jouissent d'une bonne popularité et accessibilité. Par contre, pour les circuits reconfigurables, la description en HDL s'avère bien plus complexe, imposant un plus grand degré d'expertise et des temps de développement supérieurs. Des solutions pour pallier cela sont présentées dans la section 4.2.2.

Les trois derniers points concernent plus particulièrement les architectures parallèles, soient-elles multi-processeur, mixtes ou hétérogènes. D'ailleurs, il est pertinent de faire remarquer que toute plateforme basée sur un dispositif reconfigurable constitue potentiellement une architecture parallèle, indépendamment de la présence ou non d'autres éléments de traitement. Ces architectures parallèles posent un certain nombre de problèmes lors de la description et de la programmation des applications, des problèmes bien spécifiques qui ne surgissent pas lors de l'utilisation d'une plateforme monoprocesseur. Ces aspects sont abordés dans la suite.

4.2.1 Méthodes d'implémentation et modèles de description

Depuis l'apparition des premières machines parallèles, plusieurs approches dédiées à la programmation et à la formalisation d'applications pour ces architectures ont vu le jour. En effet, la programmation d'une architecture parallèle ne se résume pas à programmer individuellement ses différents éléments de calcul et leur fonctions de traitement. Il est également nécessaire de prendre en compte :

- la communication entre les éléments de calcul,
- le partage des données sources et la fusion des données résultantes,
- la synchronisation,
- la gestion du modèle d'exécution (pipeline, Split-Compute-Merge, ..),
- le partitionnement optimal des traitements entre les éléments de calcul.

Une des approches les plus répandues pour la description d'applications parallèles est le formalisme flot de données. Ce formalisme préconise la description de l'algorithme par la décomposition de ceci en différents éléments indépendants et intercommunicants, responsables chacun par la réalisation d'une tâche ou traitement sur un ensemble défini de données provenant d'une source (capteur, mémoire, ou autre élément). La communication entre ces éléments est réalisé au moyen de l'échange de jetons (ou *tokens*), et l'exécution des différentes étapes de l'algorithme est réalisée de façon concurrente, en respectant bien évidemment les différentes relations de dépendance de données. La représentation finale de l'algorithme est faite au moyen d'un graphe flot de données.

Basé sur ce formalisme et sur la méthodologie AAA (Adéquation Algorithme Architecture), l'environnement SynDEx [80, 81], développée par l'INRIA, propose un cadre pour le développement et le partitionnement d'applications concurrentes pour des architectures distribuées hétérogènes. L'algorithme est décrit sous la forme d'un graphe orienté acyclique. Chaque noeud correspond à une opération, et chaque arc orienté correspond à une dépendance de données reliant une opération productrice de données (ou tokens), à une opération consommatrice (fig. 4.3) [82]. Procédant ainsi, la relation d'ordre entre les différentes opérations est déterminée uniquement par leur dépendance de données, permettant l'expression du parallélisme potentiel entre les différentes tâches.

La plateforme cible est également modélisée sous la forme d'un graphe orienté, où chaque noeud correspond à une machine d'état (FSM), pouvant signifier un opérateur (microprocesseur, DSP, FPGA, ...), une mémoire, un bus ou un *"communicateur"* (équivalent à un canal DMA). Les arcs du graphe représentent les connections entre les entrées et les sorties des multiples machines d'états, constituant ainsi un réseau de FSM's (fig. 4.4).

Une fois en possession d'une paire de graphes représentant l'algorithme d'une part et l'architecture de l'autre, l'obtention du graphe d'implémentation est faite au

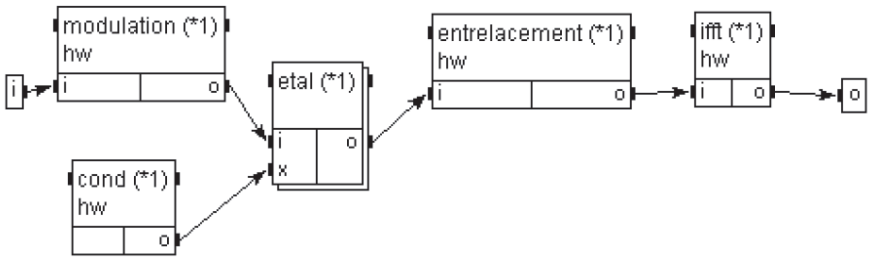


FIG. 4.3 – *Exemple de graphe d'algorithme créé avec l'outil SynDEx*

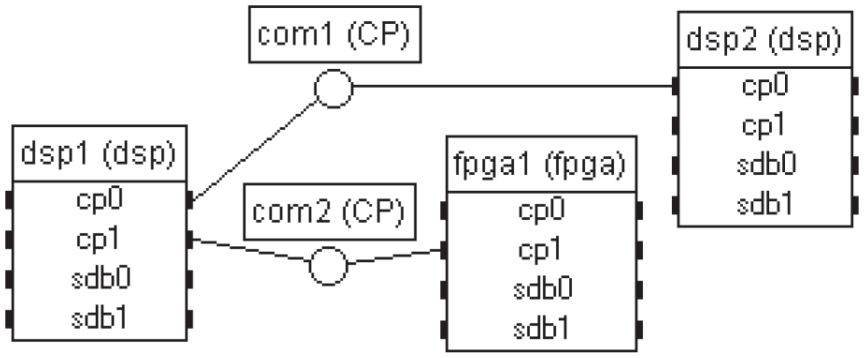


FIG. 4.4 – *Exemple de graphe d'architecture créé avec l'outil SynDEx*

moyen de la transformation du graphe d'algorithme en fonction du graphe d'architecture. Cette adéquation est faite par le *"mapping"* (allocation) et le *"scheduling"* (ordonnancement) des opérations et transferts de données du graphe algorithme, dans les opérateurs et médias de communication du graphe architecture. Ceci est réalisé par une approche heuristique, prenant en compte la durée des traitements et communications inter-composants, qui sont définies au préalable par le programmeur.

Le résultat est présenté sous la forme d'un macro-code pour chaque noeud du graphe d'architecture. Ce macro-code est composé de directives concernant les appels aux fonctions d'opération, le contrôle des interfaces de communication, l'allocation mémoire et la synchronisation et ordonnancement des tâches (threads et sémaphores). Ce macro-code peut être "traduit" vers le langage adéquat pour chaque composant de l'architecture, en remplaçant ses directives par du code contenu dans des librairies (normalement C/C++ pour les composants software, et VHDL pour les composants hardware). Le code des opérations de traitement constituant l'algorithme doit être fourni par l'utilisateur. La figure 4.5 [82] illustre la chaîne complète de design de la méthodologie SynDEx.

Le projet Ptolemy [83], de l'Université de Berkeley en Californie, s'adresse également à l'étude de la modélisation, de la simulation et du design de systèmes concur-

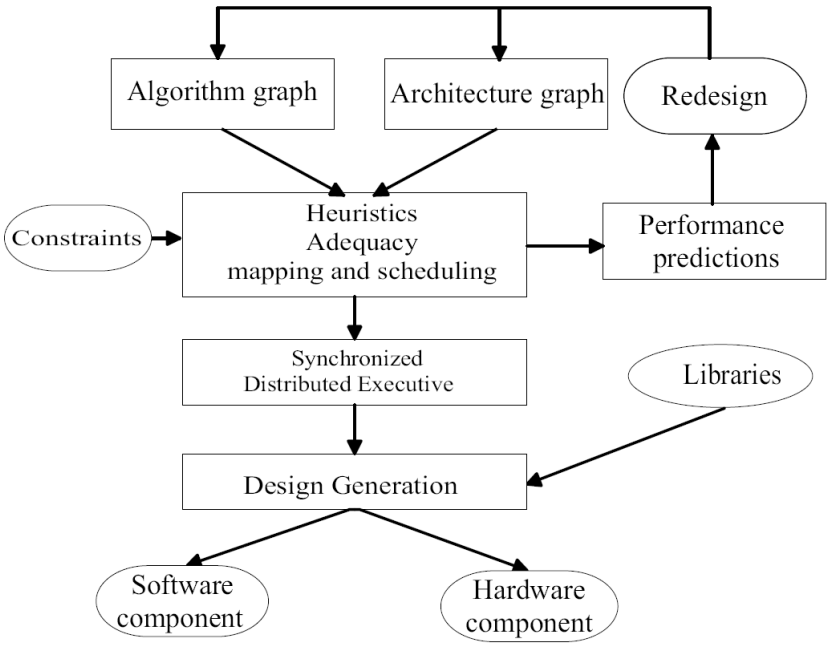


FIG. 4.5 – *Flot de conception de la méthodologie SynDEx*

rents temps-réel, avec une attention particulière aux systèmes embarqués hétérogènes. Un des résultats de ce projet est le système logiciel Ptolemy II. Il s'agit d'un système de modélisation et de simulation de systèmes hétérogènes.

Les modèles dans Ptolemy II sont construits comme un ensemble de composants inter-agissant, appelés "acteurs" (*actors*). Ceci nous emmène au concept de *actor-oriented design*, où un acteur est une entité de traitement avec des ports d'entrée et de sortie, un état interne, et des paramètres, capable d'exécuter des *actions* et de communiquer avec d'autres acteurs.

Le langage CAL [84] est un langage dédié à la description d'acteurs du type flot de données. Ce langage a également été créé dans le cadre du projet Ptolemy. Dans [85], le langage CAL a été utilisé pour la description d'un co-processeur dédié au traitement d'images. Le but de ce langage est de permettre une description concise et haut-niveau d'un acteur, en isolant le comportement de celui-ci d'une quelconque sémantique spécifique à la plateforme d'exécution.

Différents acteurs peuvent être combinés sous la forme d'un réseau afin de constituer un système plus large ou plus complexe. La connexion entre les ports d'entrée et sortie des différents acteurs est établie au moyen de canaux de communication constitués de mémoires FIFO (fig. 4.6). Dans le souci de ne pas intégrer les spécificités de la plateforme au niveau de la description de l'algorithme, ces mémoires FIFO sont considérées comme étant de taille infinie.

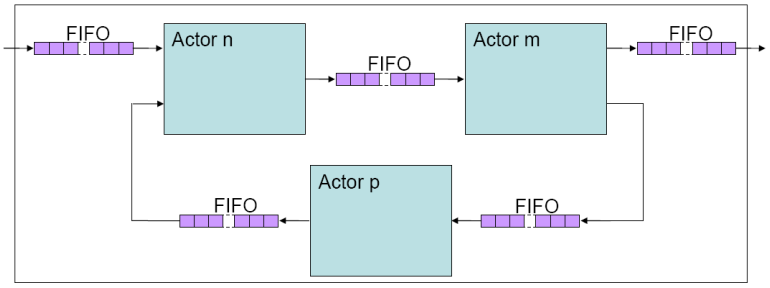


FIG. 4.6 – *Un réseau d'acteurs connectés au moyen de mémoires FIFO.*

Nous citerons également les méthodes basées sur le concept de "hardware skeletons" [86]. Un *squelette* est une abstraction paramétrable et de haut-niveau d'un modèle architectural couramment employé lors du déploiement d'applications parallèles , comme par exemple le SCM (Split-Compute-Merge) ou le *data-farming*. Ces squelettes pouvant être associés, voire imbriqués, la structure d'une application parallèle peut être exprimé par une combinaison de squelettes. Cette approche a été adaptée au contexte des circuits reconfigurables, comme il est présenté dans [87].

Le point commun entre ces différentes méthodes de description et d'implémentation d'applications parallèles est l'utilisation du formalisme flot de données. Ce formalisme se prête particulièrement bien à la description d'applications régulières de traitement de données. Néanmoins, cela ne va pas de même pour des applications irrégulières.

Par application irrégulière nous définissons les applications pour lesquelles la quantité de données à traiter, ainsi que la nature des traitements eux-mêmes, peuvent varier d'une itération à l'autre. Ceci est notamment le cas des applications de vision active et précoce auxquelles nous nous adressons.

Prenons comme exemple le réseau d'acteurs CAL présenté en figure 4.6. Pour l'implémentation effective de ce système au sein d'une plateforme, la définition de la taille des différentes mémoires FIFO mises en jeu est nécessaire. Ceci peut être réalisé de façon déterministe et off-line pour une application régulière. Par contre, dans le cas d'une application irrégulière, il est nécessaire d'appliquer une loi de type *"worst case"*. Dans notre cas cela revient à définir la plus grande taille mémoire

possiblement nécessaire pour chaque FIFO, ce qui engendrerai une allocation de ressources mémoire qui ne seraient peut-être jamais utilisées. De plus, dans certains cas, ce "worst case" ne peut même pas être calculé à priori.

Bien évidemment, il existe des stratégies pour éviter ce genre d'inconvénient (comme par exemple l'utilisation de mémoires partagées), et l'irrégularité d'une application ne rend pas prohibitive l'application du formalisme flot de données. Par contre, les applications irrégulières ne s'expriment pas "naturellement" sous ce formalisme, et nécessitent un effort supplémentaire de paramétrage afin de rendre possible leur description et leur implémentation.

D'autre part, ces méthodes s'adressent surtout aux problématiques relatives au traitement des données, et ne traitent pas les aspects relatifs au contrôle de la plateforme. Ainsi, même s'il ne faut pas les écarter définitivement, car elles peuvent s'avérer utiles pour l'implémentation de certaines applications ou parties d'application, elles ne représentent sûrement pas une réponse complète aux exigences des caméras intelligentes évoquées en début de chapitre, à savoir la description (et la programmation) de l'application **et** la gestion bas-niveau des éléments de la plateforme.

4.2.2 Approches dédiées aux systèmes reconfigurables

Un certain nombre de méthodologies et d'outils existent pour simplifier l'implémentation d'applications au sein des dispositifs reconfigurables [88], en évitant la complexité de la programmation en langage HDL.

Certaines de ces approches consistent en la "traduction" de langages de programmation haut-niveau vers un langage HDL synthétisable. Le plus souvent, le langage haut niveau utilisé est un sous-ensemble du langage C, augmenté d'un certain nombre de directives concernant le parallélisme, la synchronisation et l'échange des données entre tâches.

Une deuxième catégorie d'approches s'appuie sur l'instantiation de processeurs "soft-core" au sein du dispositif reconfigurable, permettant ainsi l'utilisation de langages de programmation haut niveau et l'exploitation des outils de compilation et développement dédiés à ces langages.

Ces deux approches seront traitées en détail dans les sections suivantes. Nous pouvons aussi citer l'extension de la méthodologie SynDEx, appelée SynDEx-IC [89, 90]. En se basant sur la chaîne de conception de SynDEx, SynDEx-IC est capable de générer du code RTL correspondant aux chemins de données et de contrôle d'une application décrite au moyen d'un graphe flot de données.

4.2.2.1 Les traducteurs *"C-like to HDL"*

Avec l'évolution et la démocratisation des dispositifs électroniques reconfigurables, il est apparu des outils permettant la traduction de code haut-niveau (ou *"C-like"*) d'une application en design HDL synthétisable.

Le langage Impulse-C, développé par Impulse Accelerated Technologies, est un sous-ensemble du langage C, accompagné d'une librairie de fonctions qui assure un support de programmation parallèle. Cette librairie contient aussi bien des fonctions que des types de données, permettant le partitionnement d'applications écrites en C sur des architectures parallèles hétérogènes, composées de processeurs standards et de logique reconfigurable.

Basé sur ce langage, l'environnement de co-design Impulse CoDeveloper [91] (fig. 4.7) propose des outils de programmation, compilation, débogage, partitionnement et co-simulation (hardware/software). L'outil est capable de produire en résultat le code HDL pour une partie de l'application, ainsi que les éléments d'interface entre les modules logiciels et matériels.

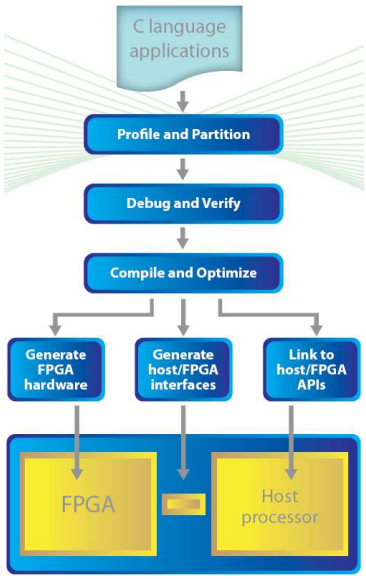


FIG. 4.7 – *Schéma descriptif de l'outil de co-design Impulse CoDeveloper.*

Le modèle de programmation est basé sur des processus qui sont appliqués sur des flux de données (streams). Les processus sont des éléments indépendants, responsables de l'exécution concurrente de différentes parties de l'application. La communication des flux entre les processus est faite au moyen de mémoires FIFO dual-clock, permettant ainsi de s'affranchir des contraintes de synchronisation. La communication par mémoire partagée est également possible. Il s'agit en effet d'un concept très proche de la notion d'*acteur*, et du formalisme flot de données.

D'autres langages et approches similaires existent. Le langage Handel-C, développé dans l'Université de Oxford [92], en est un exemple. Le kit de développement *Design Suite*, proposé par l'entreprise Agility [93], est un environnement de développement pour le prototypage rapide sur FPGA basé sur ce langage. Nous citerons également le compilateur FpgaC [94], héritier du TMCC (Transmogrifier C), de l'Université de Toronto. Il s'agit d'un outil "open source" capable de compiler un sous-ensemble du langage C pour en créer une "netlist" de description d'un circuit numérique.

Mis à part les approches basées sur le langage C, d'autres langages haut-niveau peuvent être exploités. Le compilateur MATCH (MATlab Compiler for distributed Heterogeneous computing systems) [95] permet l'implémentation d'une application décrite en langage MATlab dans une architecture hétérogène et distribuée. L'architecture cible peut donc contenir des processeurs *general purpose*, des DSP's, et des FPGA. La génération du code HDL pour la partie implémentée dans le FPGA est prise en charge par le compilateur, en faisant appel à des bibliothèques pré-implémentées contenant des fonctions telles que la multiplication de matrices, le filtrage numérique ou la DFT (Discrete Fourier Transform) (fig. 4.8).

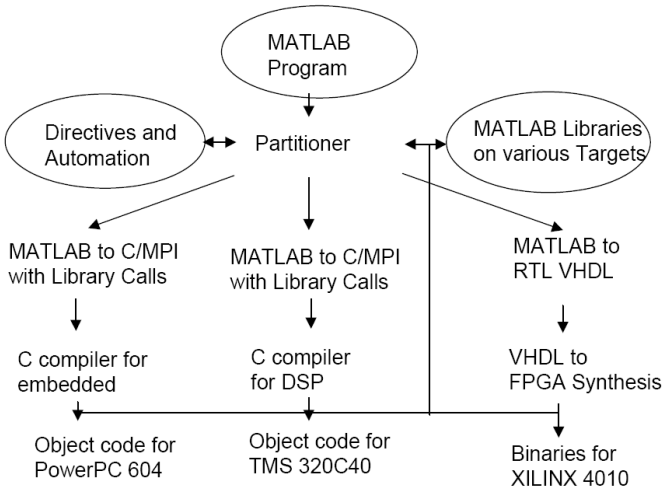


FIG. 4.8 – *Exemple de compilation du code d'une application avec le compilateur MATCH.*

Malgré l'intérêt de ce type d'approche vis-à-vis du gain en temps de développement, la traduction automatique d'un langages haut-niveau vers le HDL soulève un certain nombre d'inconvénients. L'inconvénient majeur est le fait que le code final de l'application étant généré par un automate, le développeur perd d'une certaine façon le contrôle sur son application. La compréhension, débogage, modification ou réutilisation du code final généré est très difficile, voire impossible pour les tâches les plus complexes.

Afin d'illustrer cela, l'annexe A contient le résultat de la conversion en HDL d'une application simple. L'outil utilisé est un compilateur "C to Verilog", disponible gratuitement en ligne [96]. La fonction implémentée calcule terme à terme la moyenne entre deux entiers provenant de deux tableaux d'entrée, et écrit le résultat dans un tableau de sortie. Le calcul est réalisé sur des tableaux contenant une centaine d'entiers. Alors que le code C de l'application comprend environ une dizaine de lignes, le code HDL généré occupe trois pages entières! Si cela ne remet pas en cause l'efficacité de l'outil, car le code généré est bien synthétisable et fonctionnel, il est néanmoins évident que le contrôle du programmeur sur son design est sérieusement compromis.

Si cette "perte de contrôle" peut être tout à fait tolérable ou admissible dans le cas de l'implémentation d'une fonction de traitement, la conception de l'intégralité d'un système au moyen de cette approche semble peu appropriée. Le résultat obtenu serait une sorte de "boîte noire", dont on connaît les entrées et les sorties, mais dont on ignore complètement ce qui se passe à l'intérieur. Et surtout, dont on ne peut pas estimer le temps du parcours l'entrée et la sortie.

En dernier nous citerons le langage JHDL [97], développé dans la Brigham Young University aux États-Unis. Il s'agit d'un langage de description hardware implémentée en Java, et disposant de plusieurs outils de design et simulation *open source*. L'environnement permet de décrire un circuit en langage haut-niveau à l'aide de classes Java, et est capable de produire en sortie une "netlist" compatible avec plusieurs familles de composants FPGA de chez Xilinx.

Malgré leur apparente ressemblance, cette approche est sensiblement différente de celles présentées avant, car il ne s'agit pas d'un "traducteur" de langage haut-niveau vers langage HDL. Le JHDL est lui-même un langage HDL, mais basé sur un certain nombre d'objets et classes Java afin de permettre une programmation de plus haut niveau par rapport aux langages VHDL ou Verilog. La figure 4.9 montre l'exemple de la description d'un additionneur complet 1 bit en JHDL.

```
// Import the base libraries for JHDL design
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;

// Our FullAdder is a Java class.  Begin the class declaration here
public class FullAdder extends Logic {

  // This is cell's interface (ports)
  public static CellInterface[] cell_interface = {
    in("a", 1),
    in("b", 1),
    in("cin", 1),
    out("sum", 1),
    out("cout", 1)
  };

  // This is the constructor - it gets called when a new FullAdder is desired
  public FullAdder(Node parent, Wire a, Wire b,
                   Wire cin, Wire sum, Wire cout) {

    // Since we extend Logic, always have to call its constructor as the first thing
    super(parent);

    // Connect the wires passed in as parameters to the constructor to
    // the ports from the CellInterface above.
    connect("a", a);
    connect("b", b);
    connect("cin", cin);
    connect("sum", sum);
    connect("cout", cout);

    // Build our logic as a collection of gates.
    or_o( and(a,b), and(a,cin), and(b,cin), cout ); /* cout is output */
    xor_o( a, b, cin, sum );                        /* sum is output */

  }
}
```

FIG. 4.9 – *Description d'un additionneur complet 1 bit en langage JHDL.*

4.2.2.2 Les approches basées sur les processeurs soft-core

Les approches basées sur processeurs soft-core permettent d'éviter la complexité de la programmation en langage de description matérielle, au moyen de l'instantiation d'un coeur de processeur au sein du dispositif reconfigurable. Il est alors possible d'implémenter un traitement dans le FPGA sans pour autant le programmer en HDL, mais plutôt en programmant ce coeur de processeur en langage C ou C++. Le dispositif programmable se présente sous la forme d'un processeur dit "soft-core", qui partage un grand nombre de caractéristiques avec les processeurs classiques de type RISC, en présentant l'avantage supplémentaire d'être paramétrable.

Un deuxième avantage est la possibilité "d'augmenter" le processeur, par l'ajout d'instructions spécifiques ou de modules de traitement dédiés interfaçables avec celui-ci (accélérateurs matériels).

La société Altera propose l'outil SOPC Builder, intégré dans l'environnement Quartus II, qui permet de gérer l'implémentation et la programmation des processeurs "soft-core" NIOS II (fig 4.10). La société Xilinx propose également son processeur "soft-core" MicroBlaze pour sa gamme de dispositifs FPGA (fig. 4.11).

L'avantage à utiliser ce type de processeur, outre le gain substantiel en temps de développement, est la disponibilité d'outils de programmation (compilateur pour langage haut-niveau), simulation et synthèse. De plus, ces dispositifs sont optimisés pour l'implantation dans une certaine famille de composants FPGA, ce qui assure une utilisation optimale des ressources disponibles au moment du placement/routage. Par contre, n'étant pas complètement configurables, il est possible que l'élément soit sous-utilisé par certaines applications, ce qui n'est pas souhaitable s'il existe une contrainte spatiale à respecter (occupation des ressources du FPGA). Mais l'inconvénient majeur provient du fait qu'il s'agit bien de modules IP (Intellectual Property), et donc payants, ce qui implique dans un code source fermé.

Des exemples d'utilisation de ce type de dispositif peuvent être retrouvés dans [31] et dans [98]. Dans ces deux travaux des processeurs NIOS sont utilisés comme modules de contrôle et supervision d'un système contenant plusieurs éléments de calcul. L'utilisation de ce type de processeur en tant qu'élément de calcul lui-même est également envisageable. Dans [99] une approche SoPC d'une grille de calcul est exploitée, avec un réseau configurable de processeurs MicroBlaze (NoC - Network on Chip).

D'autres entreprises proposent des approches soft-core. Par exemple, l'entreprise suédoise Mitrionics propose une plateforme de développement nommée Mitrion [100]. Cette plateforme est composée d'un processeur virtuel et de l'environnement de développement Mitrion SDK. L'approche d'implémentation est basée d'une part sur un langage "C-like" intrinsèquement parallèle appelé Mitrion C, et d'autre part sur un processeur soft-core, reconfigurable en fonction de l'application.

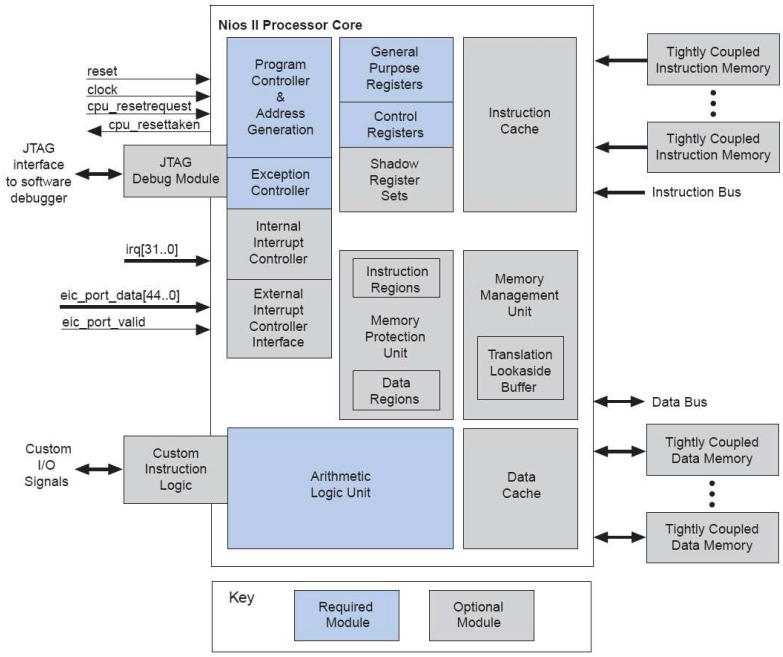


FIG. 4.10 – *Architecture interne du processeur soft-core NIOS II de chez ALTERA.*

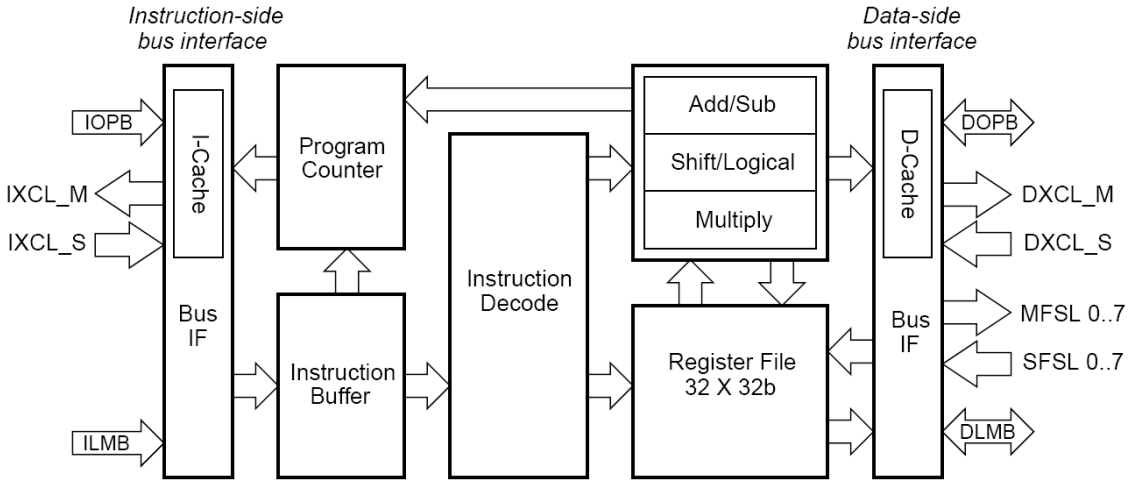


FIG. 4.11 – *Architecture interne du processeur soft-core MicroBlaze de chez XILINX.*

La méthodologie permet au programmeur de développer son programme à l'aide d'un langage adapté aux applications concurrentes. Ensuite, le code du programme est analysé, afin d'adapter l'architecture du processeur Mitrion en fonction de l'application. L'architecture résultante est finalement instanciée en tant que processeur soft-core dans le circuit FPGA.

A part l'application des approches proposés par les fabricants de FPGA ou des tiers, il est également possible de développer soi-même un processeur soft-core, taillé sur mesure pour une application donnée, ou en reprenant les caractéristiques d'un dispositif déjà existant en version COTS.

Un exemple peut être retrouvé dans [101]. Dans ces travaux, un modèle du DSP C6201 a été crée en langage VHDL. Puis, en analysant le code assembleur du programme à exécuter, les ressources matérielles nécessaires sont définies. L'architecture du DSP est alors paramétrée pour ne contenir que les ressources minimales nécessaires (nombre minimal de registres, multiplieurs, etc.). Ceci permet l'obtention d'un élément programmable optimisée en fonction de l'application, avec un gain spatial par rapport à la version complète du DSP qui aurait été sous-utilisée. Ce genre d'approche dévoile tout son intérêt lorsque l'on a l'intention de concevoir un réseau d'éléments de calcul. Le gain spatial engendré par l'optimisation permet l'intégration d'un plus grand nombre d'éléments dans le réseau, et une conséquente amélioration en performance. Dans le travail cité cette possibilité a été exploitée avec l'implantation de 4 instances du C6201, optimisées pour le filtre de Sobel, dans un circuit FPGA. L'accélération obtenue a été proche de 4, c'est à dire presque linéaire par rapport à l'implantation avec un seul DSP.

Un exemple de la création d'un soft-core taillé sur mesure est donné dans[102]. Un processeur à jeu d'instructions réduit (RISC) a été conçu afin de constituer l'architecture LAPMAM (Linear Array Processor with Multi-mode Access Memory). Il s'agit d'une architecture dédiée aux traitements d'images bas et moyen niveau, constituée d'un réseau linéaire de processeurs élémentaires. Le jeu d'instructions du processeur créé compte un nombre réduit et optimisé d'instructions câblées, du type arithmétique, logique et de communication. Une instruction câblée ne fait pas appel à un microprogramme. Elle utilise simplement un PLA (Programmable Logic Array) pour le décodage et l'exécution de l'opération souhaitée. Ceci permet que la plupart des instructions soient exécutées en une seule période d'horloge.

La création d'une telle structure peut être très intéressante, d'une part grâce aux bonnes performances obtenues, et d'autre part grâce à l'optimisation spatiale. Néanmoins, la conception intégrale d'un tel processeur requiert un temps de développement relativement élevé (quand comparé aux soft-core IP), et demande une étude minutieuse du type d'application visée afin d'en déterminer les besoins au niveau du jeu d'instructions et du chemin de données.

Il est important de souligner que, une fois de plus, les différentes approches d'implémentation dédiées aux circuits reconfigurables présentées sont essentiellement axées sur le traitement de données. Il s'agit en effet de méthodes plutôt dédiées aux systèmes de calcul ou co-traitement, mais pas spécifiquement ciblées sur des systèmes d'acquisition et traitement du type caméra intelligente. Ainsi, le contrôle et la coordination des éléments matériels périphériques, tels que le capteur d'images, ne sont pas directement pris en compte. Par contre, l'implémentation du coeur du système au sein même du dispositif FPGA, et donc à "proximité" de ces éléments matériels, laisse présager une possibilité d'évolution permettant de prendre en charge non seulement la gestion du traitement des données, mais également le contrôle de l'intégralité de la plateforme.

La méthodologie proposée dans cette thèse, et présentée dans le chapitre suivant, s'appuie sur une approche "soft-core", par l'instantiation d'un élément programmable dans le FPGA, mais en incluant les caractéristiques nécessaires afin de permettre la gestion bas-niveau des éléments périphériques, et en modélisant l'ensemble de la plateforme "smart camera" comme un seul et unique processeur.

Il s'agit d'une certaine manière de concevoir une couche fonctionnelle intermédiaire entre l'application et la plateforme, permettant à la fois de les réunir au moment de l'exécution, mais également de les isoler pendant les phases de développement.

Seconde partie

Méthodologie proposée, Implémentation et Résultats

Chapitre 5

Méthodologie proposée

De façon synthétique, une application de vision peut être décrite comme une séquence de procédés mathématiques appliqués sur des données décrivant une image ou une scène (pixels, primitives, ou descripteurs). Dans le cas des caméras intelligentes, ces procédés sont exécutés par différents circuits électroniques, au sein d'une plateforme embarquée d'acquisition/traitement.

D'une part, les procédés mathématiques décrivent comment extraire une information sur la scène ou l'environnement à partir d'une image ou séquence d'images. Ceci est communément appelé l'*algorithme*. D'autre part, les circuits électroniques doivent alimenter ces procédés en données, et exécuter les opérations qui les décrivent de façon convenable. Ces circuits composent ce que l'on appelle l'*architecture*.

L'objectif principal de cette thèse est d'apporter un certain nombre de solutions sur les étapes nécessaires pour passer de la description d'une application en tant qu'*algorithme* à sa description intermédiaire en tant que programme, et finalement à l'exécution de ce programme au sein d'une *architecture* de calcul enfouie et dédiée. **Nous nous concentrerons sur les architectures de type caméra intelligente, basées sur un dispositif reconfigurable de type FPGA, et dont tous les autres composants matériels sont connectés à celui-ci.**

L'ensemble des solutions apportées constituent une méthodologie d'implémentation pour les applications de vision précoce et embarquée. Afin de concevoir et tester une telle méthodologie d'implémentation, nous utiliserons comme architecture cible la plateforme SeeMOS, présentée dans la section 3.3.

Cette plateforme est constitué d'un nombre important de composants électroniques, notamment un FPGA, un DSP, un capteur d'images, des capteurs inertiels et des blocs mémoire RAM, plus l'électronique nécessaire à l'implémentation du protocole de communication Firewire. Chacun de ces dispositifs a ses propres besoins et règles en termes de programmation, synchronisation et contrôle. L'exploitation simultanée de ces différentes ressources est une tâche complexe, et justifie la nécessité d'une méthodologie d'implémentation adaptée. Cela justifie également l'utilisation d'une telle plateforme en tant que support de développement et expérimentation pour la méthodologie proposée.

Une des plus grandes difficultés concernant l'implémentation d'une application sur ce type de plateforme est due à la très forte hétérogénéité matérielle du système. En effet, comme il a été expliqué précédemment, chaque élément de la plateforme dispose de ses propres modalités de programmation et de contrôle :

FPGA : Le dispositif FPGA est programmé en langage de description matérielle (VHDL par exemple), et utilise l'environnement de programmation fourni par le fabricant du dispositif.

DSP : Le DSP est programmé en langage haut-niveau (C/C++), et utilise également un environnement de programmation propre.

Interface de communication : L'interface est programmée et contrôlée au moyen d'un protocole dédié. Du coté du système hôte cela se traduit par l'utilisation de fonctions implémentées dans la bibliothèque du pilote de communication. Cette bibliothèque est programmée en langage haut niveau (C++), et permet l'utilisation des environnements de développement et compilation existants (e.g. Visual C++). Par contre, du coté de la caméra, l'interface est contrôlée par un ensemble de signaux numériques, plus les bus d'envoi et réception des données. Ceci requiert une gestion bas niveau par un pilote matériel dédié.

Capteurs : Les capteurs (image et inertiels) sont commandés par un ensemble de paramètres et signaux numériques de contrôle, plus les bus de données en sortie des CAN (Convertisseur Analogique-Numérique). Le contrôle des capteurs requiert une connaissance approfondie des caractéristiques de ceux-ci, et peut être soumis à des contraintes strictes par rapport au timing et à la synchronisation de certains signaux, notamment pour le capteur d'images. Comme pour l'interface de communication, une gestion bas niveau par un pilote matériel s'impose.

Mémoires : Les mémoires sont contrôlées au moyen d'un protocole classique composé de cycles de lecture et d'écriture. Le pilotage des mémoires peut donc être réalisé de façon matérielle, ou alors par l'utilisation d'un contrôleur mémoire logiciel[1] (notamment ceux disponibles dans les librairies des processeurs soft-core et dans les DSP).

Afin de simplifier la programmation d'une telle plateforme, nous proposons une méthode pour homogénéiser le contrôle et l'inter-communication de l'ensemble de ces éléments au moyen d'un jeu d'instructions commun. L'objectif affiché est que l'intégralité du système (capteurs + unités de traitement + interfaces) puisse être vue par le programmeur comme étant un seul et unique processeur.

Par contre, avant de passer à la présentations des différents aspects de la méthodologie développée, nous ferons une brève présentation du langage SpecC. Ce dernier est utilisé dans le cadre méthodologique proposé, et certaines notions fondamentales de ce langage seront indispensables à la bonne compréhension de ce qui sera expliqué ensuite.

1. Bien évidemment, lors de l'exécution d'une application, tout devient "matériel", car il y a bien des signaux électriques au sein du silicium qui viennent matérialiser les différentes instructions. Ceci étant, l'appellation "logicielle" ou "matérielle" ici fait plutôt allusion à la méthode de description du contrôle : soit par des instructions textuelles abstraites, soit par le cheminement et la connexion explicites des signaux électriques en question.

Le langage SpecC

Le langage SpecC fait partie d'une catégorie de langages appelés SLDL, pour *System Level Design Language*.

L'objectif des langages SLDL est de permettre, avec un même langage, la description non seulement d'une application (algorithme - software), mais également de la structure responsable par l'exécution de cette application (architecture - hardware). L'utilisation de tels langages permet donc de modéliser un système de traitement de façon modulaire, et avec une granularité de description variable selon les besoins. Le niveau RTL (*Register Transfer Level*) peut être considéré comme étant le grain de modélisation le plus fin, et le niveau comportemental étant le plus "gros" grain.

Le design d'un système embarqué comprend [103]:

1. la description des interactions entre le système et l'environnement extérieur ;
2. la description de l'architecture du système ;
3. la modélisation du comportement des composants matériels et logiciels formant le système ;
4. la description des contraintes et besoins ;
5. la création d'un *test-bench* pour la simulation ;
6. la définition d'un ensemble de mesures pour estimer et comparer les performances.

La possibilité de réaliser toutes ces tâches à l'aide d'un seul et unique langage constitue un atout majeur des SLDL. En général, un tel langage doit présenter deux caractéristiques essentielles :

1. Il doit supporter la modélisation à tous les niveaux d'abstraction, du modèle comportemental sans contrainte temporelle, jusqu'au modèle RTL ajusté au cycle d'horloge près.

2. Les modèles créés doivent pouvoir être exécutés et simulés afin de valider la fonctionnalité et les contraintes du design.

Les deux langages SLDL les plus répandus sont SystemC [104] et SpecC [105].

SystemC est une plateforme de modélisation basée sur une extension du langage C++, constituée de librairies de classes et d'un noyau de simulation. Il constitue un langage haut-niveau unique pour la modélisation, l'analyse et la simulation d'un système embarqué. Il est également possible d'associer SystemC à des outils commerciaux pour la synthèse hardware.

SpecC est un sur-ensemble du langage C (tout programme C est un programme SpecC). Il ne s'agit pas, comme pour SystemC, d'un ensemble de librairies, mais d'un langage à part entière, conçu spécialement pour la spécification et le design des systèmes numériques embarqués. Le langage comprend donc toutes les instructions de l'ANSI-C, plus un ensemble de concepts et constructeurs nécessaires à la description de l'architecture [106, 107]. Ceci inclut :

- des nouveaux types de données (vecteur de bits de taille arbitraire, variables de type événement (event), etc.) ;
- les constructeurs définissant les interfaces et canaux de communication ;
- les constructeurs nécessaires pour définir la hiérarchie structurelle des différents "comportements" (*behaviors*) constituant le système.

En SpecC, la notion de *behavior* défini une représentation unique d'une fonctionnalité associée à une structure [103, 108]. Il s'agit en quelque sorte de l'unification de la notion d'*acteur* du langage CAL avec la notion de *hardware skeleton* (présentés en section 4.2.1). Ainsi, le comportement et la structure fonctionnelle du système sont définis par une hiérarchie de *behaviors*. La figure 5.1 illustre ceci dans le cas d'un système composé de deux *behaviors* séquentiels ($B1$ et $B2B3$), dont le deuxième *behavior* est lui-même composé de deux *sous-behaviors* en parallèle ($B2$ et $B3$) [108]. La figure 5.2 illustre différents façons de hiérarchiser les *behaviors* : machine d'états (FSM), exécution concurrente, séquentielle ou pipelinée.

Nous soulignerons également que SpecC supporte aussi bien l'ordonnancement statique des tâches (par l'hiérarchisation des *behaviors*), qu'un ordonnancement dynamique à l'aide des variables de type "event", capables de déclencher une action (comme les signaux d'une "sensitivity list" en langage HDL).

Le langage SpecC a été utilisé dans le cadre de cette thèse pour la modélisation et la simulation d'un processeur "soft-core" paramétrable, à être instancié dans une plateforme Smart Camera basée sur un composant FPGA.

Ce langage a été choisi, au détriment du langage SystemC, en raison de sa complétude et de sa syntaxe plus "légère" et dépouillée, en comparaison avec la programmation orientée objet.

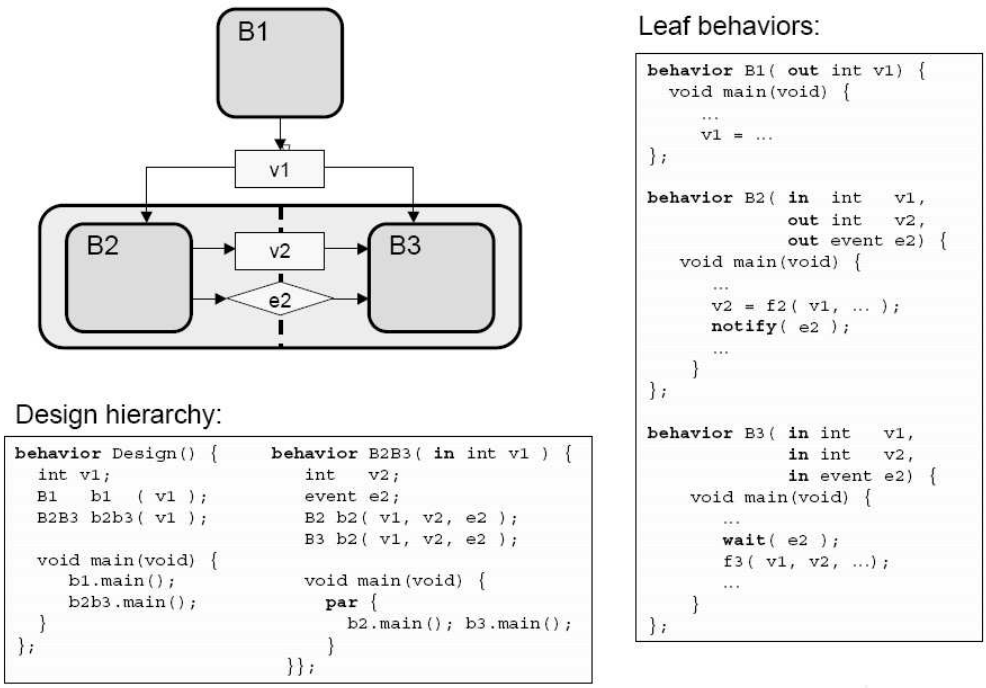


FIG. 5.1 – *Exemple de la spécification d'un système à l'aide du langage SpecC.*

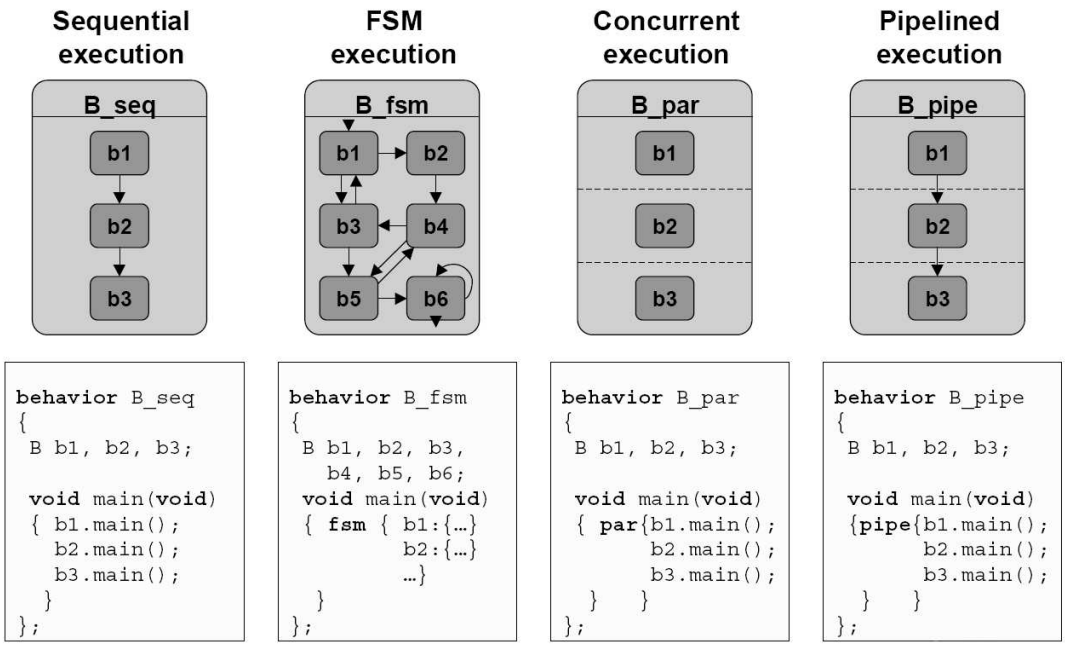


FIG. 5.2 – *Différentes hiérarchies de* behaviors *supportées par le langage SpecC.*

5.1 Description de la Méthodologie proposée

La méthodologie que nous proposons est composée des éléments suivants :

- Un langage de programmation du type assembleur, reposant sur un jeu d'instructions réduit ;
- Un outil *assembleur*, capable de transformer le code textuel du programme assembleur en code machine binaire ;
- Un modèle virtuel d'une architecture de processeur RISC, capable de décoder et exécuter les instructions assembleur. Ce modèle est écrit en SpecC ;
- Une version "soft-core" de ce même processeur, décrit en langage VHDL.

Les quatre éléments cités ci-dessus sont complètement indépendants de l'application et de la plateforme cible d'implémentation. Ils constituent le socle de base de la méthodologie, sur lequel viendront se greffer d'autres éléments afin d'adapter le système à une plateforme et/ou à une application donnée.

L'implémentation d'une **application** se fait par l'utilisation d'un certain nombre de **Processing Elements**, ou **PE's**. Ces PE's prennent en charge les opérations de traitement de données, et ils existent sous deux formes :

- PE sous forme d'un *behavior* SpecC implémentant une fonction de traitement ;
- PE sous forme de bloc fonctionnel : module de traitement programmé en VHDL, IP provenant d'une librairie, routine en C exécutée par le DSP, etc. Dans le cas du DSP, c'est le dispositif en lui-même qui est considéré comme un hyper-PE programmable.

Finalement, l'adaptation de la méthodologie vis-à-vis d'une **plateforme** donnée est faite par l'ajout des éléments suivants :

- Un ensemble de *behaviors* SpecC émulant les différents éléments matériels de la plateforme cible (capteurs, interfaces de communication, blocs mémoire, etc.) ;
- Un ensemble de pilotes câblés, programmés en VHDL et capables d'assurer la gestion de ces éléments matériels sous la coordination du processeur "soft-core".

5.1.1 Un design à deux niveaux

Le design selon la méthodologie proposée se fait à deux niveaux. Le premier niveau est composé de l'**association logicielle** du modèle virtuel du processeur avec les *behaviors* représentant les PE's, et les *behaviors* émulant les éléments de la plateforme.

Il s'agit d'un modèle virtuel complet, entièrement décrit en langage SpecC, et pouvant être utilisé pour l'exploration architecturale et algorithmique du système (e.g. partitionnement, parallélisation). Ce modèle virtuel permet la simulation, le paramétrage, et le débogage de l'ensemble "architecture + application".

Dans ce premier niveau on peut se servir d'environnements de programmation tels que *Dev-C++* ou *Code::blocks*, associés au compilateur *scrc* (SpecC Reference Compiler).

Le deuxième niveau du design est composé de l'**interconnexion** de la version soft-core du processeur avec les pilotes câblés assurant la gestion des éléments matériels, et les PE sous forme de blocs fonctionnels.

Ce deuxième niveau correspond à la version "réelle" du système (par opposition à la version "virtuelle"). Les différents choix de design découlant de l'exploration effectuée au premier niveau sont re-appliqués ici, afin de paramétrer et adapter l'architecture finale du système. L'objectif final est d'obtenir une version synthétisée et prête à être instanciée dans le FPGA.

Dans notre cas (FPGA de chez ALTERA), l'environnement Quartus est utilisé.

Le langage assembleur (et le code binaire associé) est commun aux deux niveaux, et permet ainsi de réaliser le pont entre le niveau "virtuel" d'exploration et de simulation et le niveau d'implémentation "réelle".

Nous pouvons dire que le premier niveau constitue une approche off-line, alors que le second permet d'obtenir la version temps-réel de l'application. Cette approche à deux niveaux se justifie essentiellement par les difficultés associées à l'exploration, débogage et réglage ("tuning") des paramètres du système directement au niveau VHDL. Ces difficultés proviennent d'une part du processus de compilation et synthèse VHDL, et d'autre part du langage VHDL en lui-même.

La compilation VHDL peut s'avérer assez longue, notamment pour les designs complexes. Or, lors de l'étape d'exploration architecturale du système, le design doit être compilé de nombreuses fois, afin de tester/valider les différents choix architecturaux. La compilation plus rapide du langage SpecC permet d'itérer beaucoup plus rapidement et donc de "fluidifier" ce processus. Également, lors des phases de simulation, le langage SpecC permet la création de testbenchs beaucoup plus facilement, grâce à sa syntaxe *C-like* de haut-niveau.

La granularité variable du langage SpecC permet de modéliser autant le fonctionnement du coeur du processeur au niveau RTL, que les fonctionnalités de la plateforme matérielle au niveau comportemental. D'une part, la modélisation RTL du coeur du processeur permet une simulation précise de son fonctionnement, et une "traduction" simplifiée du modèle "virtuel" vers la version HDL synthétisable du processeur "soft-core". D'autre part, la modélisation comportementale des fonctionnalités de la plateforme permet l'abstraction des particularités physiques de la caméra.

La cohabitation de ces deux niveaux d'abstraction au sein d'une même description système est fondamentale, et permet la simulation de l'architecture de façon réaliste et flexible, tout en restant détaché des aspects matériels intrinsèques à la plateforme d'implémentation.

5.1.2 Flot d'implémentation

Le flot d'implémentation se déroule de la façon suivante (fig. 5.3):

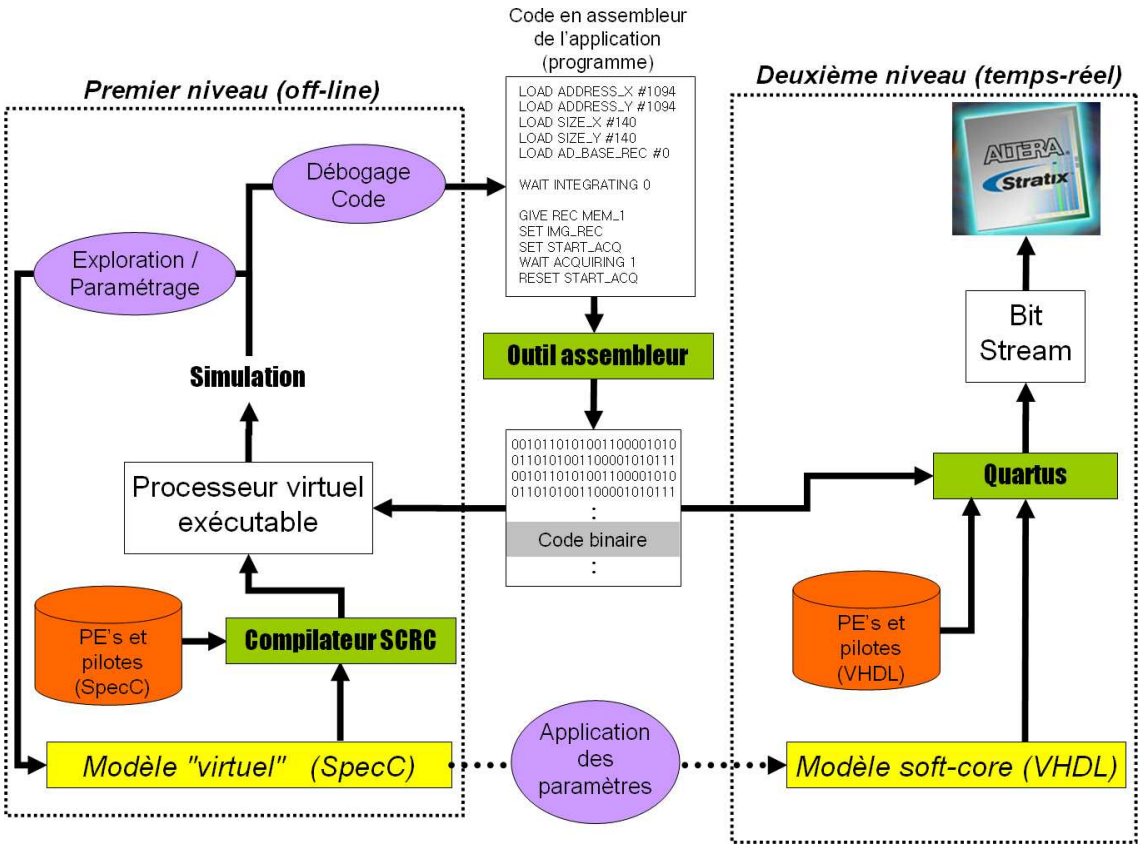


FIG. 5.3 – *Schéma du flot d'implémentation de la méthodologie proposée.*

La première étape consiste à programmer l'application à l'aide du langage assembleur. Le principal travail consiste à scinder l'application en deux parties : une partie contrôle, et une partie traitement. Ceci vient simplifier la tâche de programmation, car les fonctions de traitement pouvant être prises en charge par les PE's, le programme en assembleur peut ne contenir que les informations de contrôle de l'application. Ainsi, le programme décrit essentiellement l'ordonnancement des tâches, leur dépendance de données et l'allocation des ressources mémoires aux différents opérateurs (PE's et pilotes). Le jeu d'instructions composant ce langage, ainsi que l'outil assembleur sont expliqués en détail dans la section 5.3.

Premier niveau : design off-line

Une fois en possession du code binaire de l'application, celui-ci peut être exécuté en simulation par la version virtuelle du processeur écrite en SpecC. Cette version virtuelle peut être adaptée en fonction de la plateforme cible, par l'inclusion de *behaviors* SpecC émulant les structures matérielles de la plateforme. Elle peut également être adaptée en fonction de l'application, par l'inclusion de *behaviors* représentant les PE's de traitement de données.

L'inclusion de ces *behaviors* se fait d'une part par leur déclaration dans le corps du programme principal SpecC, et d'autre part par l'affectation de leurs ports d'entrée et sortie aux ports adéquats de l'architecture. Ces *behaviors* peuvent soit exister au préalable au sein d'une bibliothèque, soit être créés par le programmeur en langage C ou SpecC (une fonction C peut être facilement encapsulée dans un *behavior* SpecC).

Comme il a déjà été commenté auparavant, la granularité de description des *behaviors* est variable. Cela peut aller d'un module générique simulant le fonctionnement d'un capteur d'images (fonction C qui copie une image du disque dur dans une zone mémoire spécifiée), jusqu'au module RTL émulant précisément (au coup d'horloge près) le fonctionnement d'un modèle de capteur spécifique. Le premier permet un développement rapide pour une simulation fonctionnelle, tandis que le deuxième permettra une simulation très précise au niveau temporel, au prix d'un développement plus long.

Les "Processing Elements" sont expliqués plus en détails dans la section 5.4. Les pilotes hardware sont abordés dans le chapitre suivant, en prenant comme exemple la plateforme SeeMOS.

Le modèle SpecC, après inclusion des *behaviors* adéquats, est compilé par le compilateur *SCRC*. Le résultat obtenu est une version exécutable du modèle virtuel du processeur, capable d'exécuter les instructions du programme d'application. En fonction des résultats de simulation, le programmeur pourra :

- déboguer ou optimiser le code d'application, en agissant sur le programme.
- explorer ou optimiser l'architecture, en agissant sur le modèle SpecC.

Deuxième niveau : implémentation temps-réel

Une fois le modèle SpecC validé, les paramètres retenus sont appliqués dans le modèle soft-core (VHDL) du processeur. Ces paramètres peuvent concerner la structure interne du processeur (nombre de registres, taille de la pile), l'adaptation à la plateforme (déclaration et connexion des pilotes câblés), ou l'adaptation à l'application (déclaration et connexion des PE's). Ceci est fait à l'aide de l'environnement graphique Quartus[2]. Le modèle VHDL est finalement compilé et synthétisé, en faisant appel aux librairies adéquates contenant le code des pilotes et PE's utilisés. Le code binaire de l'application est également inclus à la compilation, et sert à initialiser la mémoire programme du processeur.

Le résultat obtenu est un fichier *bit stream* de configuration du FPGA. Le chargement de ce fichier dans le dispositif permettra enfin l'exécution temps-réel de l'application sur la plateforme.

En résumé :

1. nous proposons de diviser l'application en partie contrôle et partie traitement ;

2. la partie contrôle est programmée à l'aide d'un jeu réduit et simple d'instructions assembleur ;

3. la partie traitement est prise en charge par un ensemble de PE's, qui peuvent être développées de façon indépendante par rapport au reste du système (pas de contrainte de synchronisation) ;

4. l'ensemble "contrôle + traitement" peut être simulé à l'aide d'un modèle virtuel, écrit en SpecC et adapté à la plateforme cible, plus facile à modifier et paramétrer que son équivalent en VHDL ;

5. le design en VHDL n'est réalisé qu'à l'itération finale, par l'application d'un certain nombre de paramètres et modifications au modèle VHDL l'architecture. Ces paramètres et modifications émergeront de l'étape précédente de simulation à l'aide du modèle virtuel ;

6. l'implémentation temps-réel de l'application est obtenue par la synthèse du design VHDL, et son instantiation dans un circuit FPGA.

2. Il est important de souligner que l'approche proposée est parfaitement indépendante du fabricant du dispositif FPGA. Ainsi, l'environnement Quartus peut être remplacé par un autre environnement de design et synthèse, selon le fabricant du dispositif FPGA intégré dans la plateforme cible.

5.2 Architecture du processeur

Le processeur doit répondre à deux exigences fondamentales :

– Gérer le fonctionnement en parallèle des divers modules d'une caméra intelligente ;

– Permettre une gestion souple (adaptable et réactive) du parcours des données entre les capteurs, les traitements et l'interface de communication.

Pour cela nous proposons une architecture basée sur un coeur de processeur de type RISC [109], et augmentée de certaines fonctionnalités afin de répondre aux exigences sus-citées. Nous rappelons que ces exigences (parallélisme et adaptabilité) sont en rapport direct avec le cadre d'application de la vision précoce, comme présenté précédemment dans le chapitre 2.

L'architecture repose sur trois parties fondamentales :

1. un coeur de processeur du type RISC ;

2. un ensemble de modules de contrôle hardware (pilotes ou *drivers*) et de traitement de données (PE's) ;

3. un ensemble de blocs mémoire reliés à un dispositif **Crossbar**, qui permet leur exploitation en parallèle, ainsi que leur re-allocation dynamique aux différents pilotes et PE's.

La figure 5.4 illustre un synoptique général de l'architecture système proposée. Le coeur du processeur, ou *Central Control Unit* (CCU) dans le schéma, est la partie permettant le décodage et l'exécution des instructions contenues dans le programme. La CCU commande le restant du système, et assure le bon déroulement de l'application selon ce qui a été décrit par le programmeur.

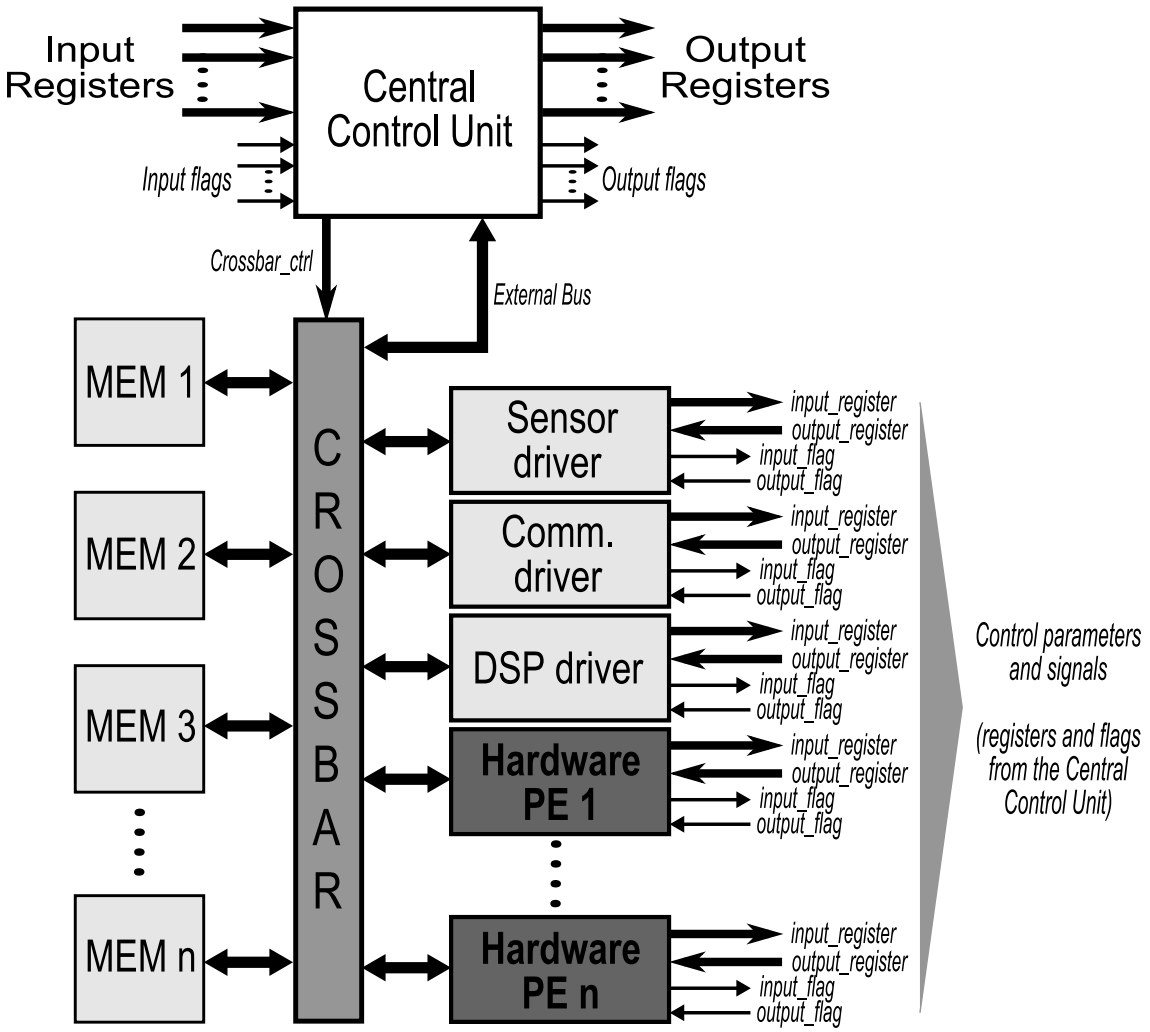


FIG. 5.4 – *Schéma synoptique décrivant l'architecture générale du système.*

5.2.1 L'unité centrale de contrôle (CCU)

L'unité centrale de contrôle est composée de quatre blocs inter-communicants (fig. 5.5) : le décodeur d'instructions, un bloc responsable du contrôle du programme, un bloc responsable du contrôle des données, et un bloc responsable des opérations sur les données (ALU).

Les instructions sont composées d'un opcode de 8 bits, et de deux opérandes de taille égale à n bits. La valeur n est en effet la "taille" en bits du processeur, et déterminera donc la largeur des registres, des bus de données et adresse, ainsi que des opérations effectuées dans l'ALU. L'architecture est adaptable à différentes valeurs de n (8, 16, 24, 32, ...). Pour la suite de ce manuscrit nous avons retenu $n = 32$, et on peut donc parler d'une architecture RISC 32 bits.

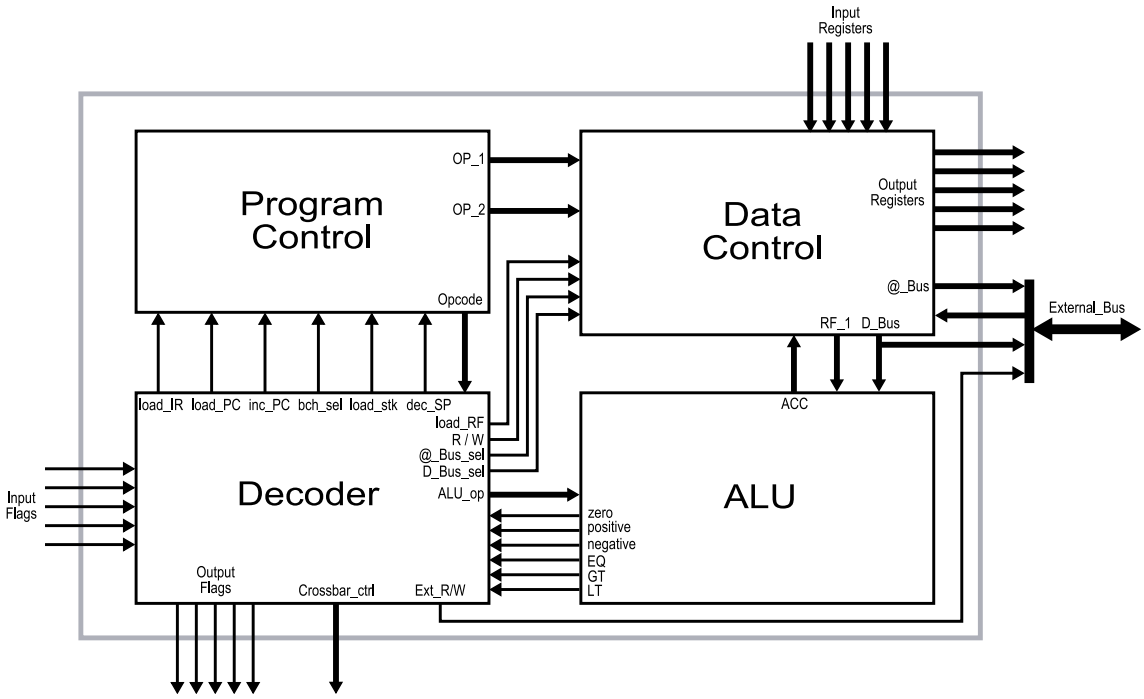


FIG. 5.5 – *Schéma synoptique du coeur du processeur (Central Control Unit).*

L'exécution de toutes les instructions est réalisé en trois cycles d'horloge:

1. 1^{er} cycle → fetch : chargement du registre d'instruction et incrément du compteur programme ;

2. $2^{ème}$ cycle → decode : décodage et définition du data-path selon l'opcode reçu ;

3. $3^{ème}$ cycle → execute : exécution proprement dite de l'instruction (chargement de registre, écriture en mémoire, mise à 1 ou zéro de flag, etc.).

Opcode	**Opérande 1 (OP_1)**	**Opérande 2 (OP_2)**
8 bits	*32 bits*	*32 bits*

TAB. 5.1 – *Format des instructions.*

Le décodeur d'instructions est composé d'une machine à états finis, permettant d'implémenter ces trois cycles pour chacune des instructions. Il est chargé de la génération de tous les signaux de contrôle destinés aux autres blocs, de la définition du mot de contrôle pour le crossbar, de la mise à 1 ou zéro des flags de sortie, et de la réception des différents flags provenant de l'extérieur (input flags) et de l'ALU. Le déroulement de certaines instructions est conditionné à l'état de ces flags (e.g.

les branchements conditionnels). Les différentes instructions implémentées par le décodeur sont présentées dans la section 5.3.

5.2.1.1 La vocation de la CCU

Avant de continuer avec la description de l'architecture interne de l'unité centrale de contrôle, il est très important d'expliquer clairement quelle est sa vocation. Même si l'architecture présentée dispose d'unités capables de traiter les données (ALU et multiplieur), la vocation première de la CCU n'est pas de faire du traitement des signaux proprement dit. Ces unités de traitement sont présentes pour permettre la gestion des données relatives au contrôle de l'application, comme par exemple l'incrémentation des compteurs de boucle, ou le calcul du nombre de pixels d'une image d'après son nombre de lignes et colonnes. La CCU, comme son nom l'indique bien, est une unité centrale de **contrôle**.

Même s'il est possible de faire du traitement des signaux avec la CCU, car les instructions et unités hardware nécessaires sont bien présentes, ce n'est pas cela que nous prônons dans notre approche. Comme expliqué précédemment, l'idée est plutôt de centraliser le contrôle de l'application dans la CCU, et de dispatcher le traitement des données dans des PE's dédiés. Ainsi, il est possible de tirer profit au mieux du parallélisme offert par le dispositif FPGA. Parallélisme de données d'une part, au sein des PE's, et parallélisme de tâches d'autre part, par le fonctionnement concurrent de 2 PE's ou plus.

Bien évidemment, il n'est pas nécessaire de créer un PE pour calculer, par exemple, la moyenne de deux valeurs si jamais cela est nécessaire dans une application. Ces "petites tâches" peuvent parfaitement être prises en compte par la CCU. Par contre, appliquer un traitement pixel par pixel sur toute l'étendue d'une image, en utilisant uniquement la CCU, est tout simplement rédhibitoire. Nous serions en train de passer à coté du parallélisme potentiel proposé par le FPGA, pour nous résumer à une exécution séquentielle et à faible fréquence d'horloge (les fréquences exploitables en FPGA sont nettement inférieures à celles des processeurs ASIC intégrés).

5.2.1.2 Le contrôle du programme

La figure 5.6 illustre le bloc responsable du contrôle programme. Ce bloc est composé de la mémoire programme, qui stocke les différentes instructions à exécuter, du compteur programme (program counter ou PC), du registre d'instructions et de la pile.

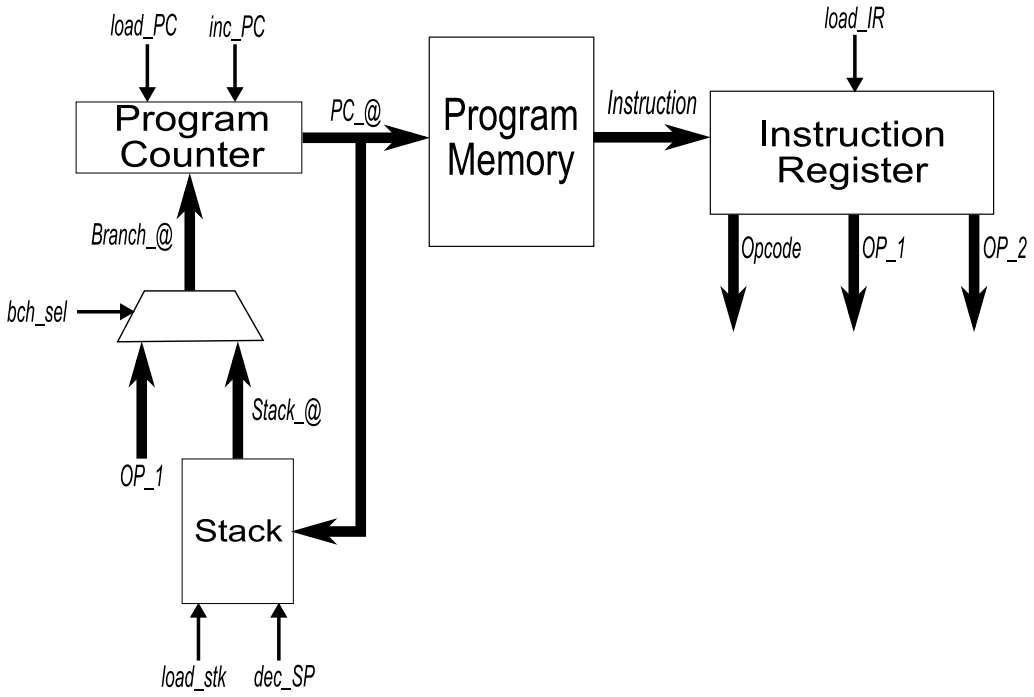


FIG. 5.6 – *Partie de l'architecture responsable par le contrôle du programme (fetching des instructions et opérations de branchement).*

Lors du premier cycle de chaque instruction, le registre d'instruction est chargé avec le contenu de la mémoire programme indiqué par l'adresse stockée dans le PC. Ce registre reçoit l'instruction entière (opcode + 2 opérandes). En sortie, les différentes parties de l'instruction sont disponibles séparément, l'opcode étant envoyé au décodeur, et les opérandes au bloc de contrôle des données.

S'il s'agit d'une instruction de branchement (inconditionnel, conditionnel dont la condition est remplie, ou appel de routine), le PC est chargé avec l'adresse contenue dans l'opérande 1. Dans le cas d'un appel de routine, la pile est chargée avec le contenu actuel du PC, avant son chargement, et le compteur de pile (stack pointer) est incrémenté. Pour effectuer un retour de routine, le PC reçoit la dernière valeur écrite dans la pile, et le compteur de pile est décrémenté.

5.2.1.3 Le contrôle et stockage des données

Les opérandes 1 et 2 sont reçues par le bloc de contrôle des données. Ce bloc est composé de la mémoire de données, du fichier registre (Register File), et des multiplexeurs de contrôle des bus d'adresses et de données. Il est illustré dans la figure 5.7.

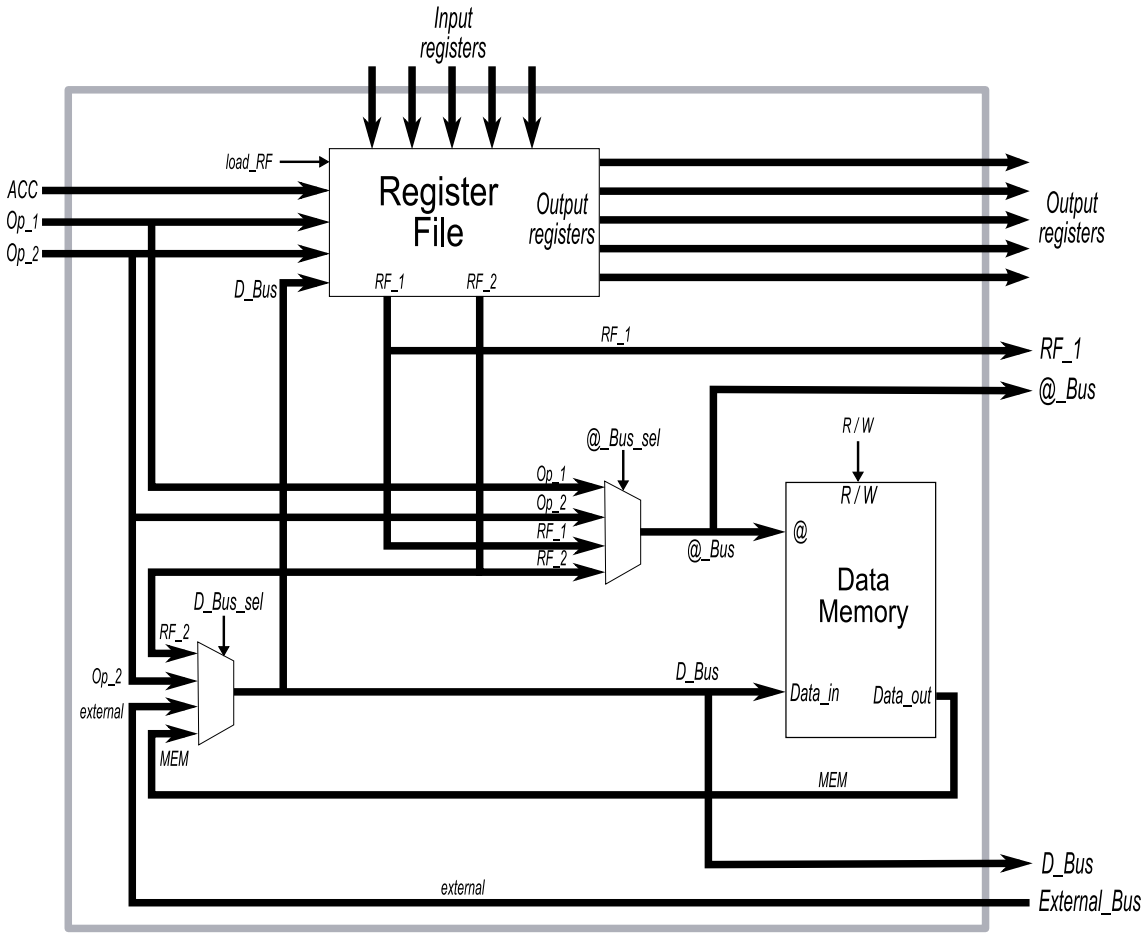


FIG. 5.7 – *Partie de l'architecture responsable du contrôle et du stockage des données.*

Le fichier registre dispose de quatre types de registres différents. Leur nombre n'est pas fixe, et peut être adapté en fonction de l'application et de l'architecture. Ainsi, si un nouveau PE est ajouté, et celui-ci a besoin, par exemple, de recevoir trois paramètres afin de configurer son fonctionnement, il est possible d'ajouter au fichier registre trois nouveaux registres de sortie pour communiquer avec ce PE.

Registres de sortie, ou write only Ces registres servent essentiellement à passer des paramètres vers les éléments périphériques de traitement, acquisition et communication. Les types de paramètres pouvant être communiqués sont par exemple la taille de l'image à acquérir, le temps d'intégration du capteur, l'adresse de base dans une mémoire pour la lecture/écriture lors d'un traitement, ou encore le nombre d'éléments à transmettre vers le système hôte lors d'une opération d'envoi.

Registres d'entrée, ou read only Ces registres sont dédiés à la réception de résultats ou status de la part des éléments externes. Par exemple, un élément de

traitement dédiée au calcul de la valeur moyenne d'une image peut communiquer son résultat à l'unité de contrôle directement par l'écriture d'un registre, plutôt que par une écriture en mémoire suivie d'un cycle de lecture de la part de l'unité de contrôle.

Registres de données Ces registres servent à stocker des données relatives au programme (variables).

Registres d'adresse Ces registres contiennent une valeur relative à une adresse mémoire, permettant l'utilisation de variables du type pointeur.

D'un point de vue fonctionnel, les registres d'adresse et données sont identiques. Il est en effet possible d'utiliser un registre de données comme registre d'adresse, et vice-versa. Leur séparation en deux catégories est plutôt destinée à aider le programmeur à mieux organiser les variables de son programme.

L'architecture interne du fichier registre est semblable à celle d'une mémoire à double-port de lecture. Le contenu du registre indiqué par l'opérande 1 est disponible sur le port de sortie **RF_1**, et celui indiqué par l'opérande 2 est disponible sur le port **RF_2**. Dans le cas d'une instruction de chargement de registre (signal *load_RF* actif), le registre indiqué par l'opérande 1 est chargé avec le contenu du bus de données. L'exception est faite pour les registres d'entrée[3], qui ne peuvent pas être modifiés par la CCU. Le signal **ACC**, provenant de l'accumulateur de l'ALU, est interprété comme étant un registre d'entrée supplémentaire.

Le contenu de chaque registre de sortie est disponible sur un port de sortie séparé, afin d'être acheminé vers les différents opérateurs intégrés à l'architecture. Les bus d'adresse et données sont également acheminés vers un bus externe, qui peut être connecté au dispositif crossbar afin de permettre l'accès de la CCU aux blocs mémoire externes à celle-ci.

Cette structure à double-port de lecture du fichier registre permet la réalisation d'opérations arithmétiques ou logiques entre deux registres au moyen d'une seule instruction (au lieu de charger l'accumulateur dans un premier temps, et puis effectuer l'opération). La sortie **RF_1** est envoyée directement vers l'ALU, et la sortie **RF_2** est acheminée vers celle-ci en passant par le bus de données.

La mémoire de données est d'usage exclusif de la CCU, et ne peut pas être utilisée par les opérateurs externes. Ceux-ci utilisent les mémoires externes accessibles par le crossbar. Étant utilisée uniquement pour stocker des valeurs et informations relatives

3. L'appellation "registre" d'entrée est en effet un abus de langage, car il s'agit en réalité de signaux d'entrée. Ces "registres" ne sont pas physiquement présents dans le fichier registre, mais plutôt dans le PE ou pilote qui les contrôlent. C'est donc uniquement leur contenu qui est visible par le fichier registre.

au contrôle du programme, sa taille peut être réduite (quelques kilo-octets). Cette taille peut être paramétrée au moment de l'implantation en VHDL du système, afin d'optimiser l'occupation des ressources.

La définition du contenu des bus d'adresse et données est faite lors du deuxième cycle d'exécution des instructions. Ainsi, le bus de données peut recevoir soit l'opérande 2 (mode d'adressage immédiat), soit le contenu d'un registre, soit une donnée lue en mémoire (mémoire données ou mémoire externe). Le bus d'adresse peut recevoir soit une opérande (mode d'adressage direct), soit le contenu d'un registre d'adresse (mode d'adressage indirect).

5.2.1.4 L'ALU

La sortie **RF_1** et le bus de données sont envoyés à l'ALU. Celle-ci est composée d'un multiplieur, d'un comparateur, et d'une unité logique et arithmétique (l'ALU proprement dite). La figure 5.8 en montre un schéma synoptique. L'accumulateur stocke le résultat de l'opération, qui est directement accessible par le fichier registre.

Les différents blocs de l'ALU opèrent sur des **nombres entiers signés** (complément de 2), de taille 32 bits.

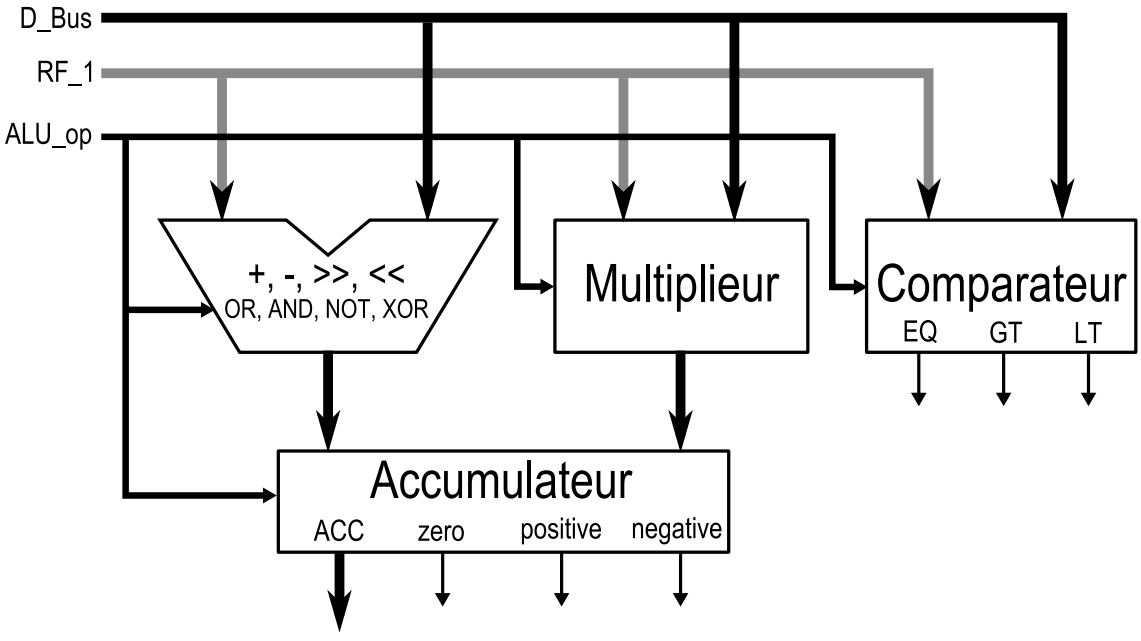


FIG. 5.8 – *Schéma de l'ALU (Arithmetic Logic Unit).*

Le comparateur compare le contenu de **RF_1** avec celui du bus de données, et gère les flags `EQ` (EQual), `GT` (Greater Than) et `LT` (Less Than) en conséquence. Ainsi :

- si **RF_1** = **D_Bus**, `EQ` = 1, sinon `EQ` = 0;
- si **RF_1** > **D_Bus**, `GT` = 1, sinon `GT` = 0;
- si **RF_1** < **D_Bus**, `LT` = 1, sinon `LT` = 0;

L'unité logique et arithmétique est capable de réaliser des additions et des soustractions, ainsi que les opérations logiques `and`, `or`, `not` et `xor`, et des *bit-shifts* à droite et à gauche. L'opération à effectuer est déterminée par le signal **ALU_op** provenant du décodeur. L'accumulateur est chargé avec le résultat de la dernière opération réalisée, et les flags `zero`, `positive` et `negative` décrivent l'état de son contenu. Les flags du comparateur et de l'accumulateur sont envoyés au décodeur d'instructions. Ces informations seront utilisées afin de déterminer la réalisation ou non des sauts conditionnels.

5.2.2 Le dispositif Crossbar

L'allocation des ressources mémoire de la plateforme aux différents modules de contrôle et traitement est faite par la reconfiguration du *Crossbar*. Ceci s'avère être un élément essentiel de l'architecture proposée, car grâce à celui-ci il est possible d'exploiter simultanément plusieurs blocs mémoires, ainsi que de redéfinir le chemin de données de façon dynamique, sans faire appel à une reconfiguration du FPGA.

Étant asynchrone, le fonctionnement du dispositif crossbar est indépendant des contraintes temporelles des opérateurs qui lui sont reliés, et qui peuvent donc fonctionner à des cadences différentes. En outre, il peut être utilisé aussi bien pour connecter des mémoires externes que des mémoires internes au composant FPGA.

La configuration du crossbar est définie par un mot de contrôle qui indique quel port d'entrée est connecté a quel bloc mémoire. De cette façon, il est possible d'exploiter facilement des stratégies telles que le "memory swapping", tout comme l'exploitation simultanée de plusieurs blocs mémoire par différents PE's. Cette exploitation simultanée est impossible avec l'utilisation d'un bus partagé entre les différents éléments, sans citer la complexité du système d'arbitrage du bus.

Prenons pour exemple une plateforme composée de trois blocs mémoire indépendants, et une application composée d'une étape d'acquisition d'image, une étape de traitement sur cette image, et une étape d'envoi des données résultantes vers le système hôte. Dans un premier temps, la mémoire 1 est allouée au capteur d'images (ou plutôt à son pilote), et une image est acquise et stockée dans cette mémoire. Dans le cas d'un système à bus partagé, le contrôle du bus est donné ensuite au PE responsable du traitement, lors de la deuxième étape de l'application. Or, pendant

ce temps, une nouvelle acquisition d'image est impossible, car le bus est occupé. Dans le cas d'un système à mémoire distribuée (une mémoire pour le capteur, et une pour le PE), il est nécessaire de copier le contenu d'une mémoire vers l'autre.

Avec l'utilisation du crossbar, il est possible de ré-allouer la mémoire 1 au PE, et d'allouer la mémoire 2 au capteur. Ainsi les deux éléments peuvent travailler en parallèle. Les accès mémoire se font de façon parfaitement indépendante, sans arbitre de bus, latence, ou copie mémoire. Du point de vue du capteur ou du PE, cette mémoire lui appartient, comme dans un système à mémoire distribuée. Par contre, du point de vue du système, le PE et le capteur agissent bien sur la même mémoire, comme ci celle-ci était partagée. L'application peut donc se dérouler comme indiqué ci-dessous, et illustré dans la figure 5.9.

1. Acquisition d'une image en mémoire 1 ;
2. Traitements des données en mémoire 1, et acquisition d'une image en mémoire 2 ;
3. Envoi des données en mémoire 1 vers le système hôte, traitement des données en mémoire 2 et acquisition d'une image en mémoire 3 ;
4. Envoi des données en mémoire 2 vers le système hôte, traitement des données en mémoire 3 et acquisition d'une image en mémoire 1 ;
5. Envoi des données en mémoire 3 vers le système hôte, traitement des données en mémoire 1 et acquisition d'une image en mémoire 2 ;
6. Retour à l'étape 3.

Par souci de simplicité, la figure 5.9 considère que l'acquisition, le traitement et l'envoi prennent tous le même temps d'exécution. Ceci est rarement le cas dans une application réelle, mais ne représente aucun problème vis-à-vis de l'architecture, car l'unité de contrôle dispose des instructions nécessaires afin de synchroniser les différentes étapes.

Avec une approche séquentielle classique (bus partagé), la latence de l'application décrite serait donnée par la somme des temps d'acquisition, traitement et transmission. Dans une approche à mémoire distribuée, il serait nécessaire de comptabiliser le surcoût lié à la copie des données entre les mémoires. Avec l'approche proposée, les différentes étapes sont pipelinées, et la cadence de l'application sera égale à la cadence de l'étape la plus lente.

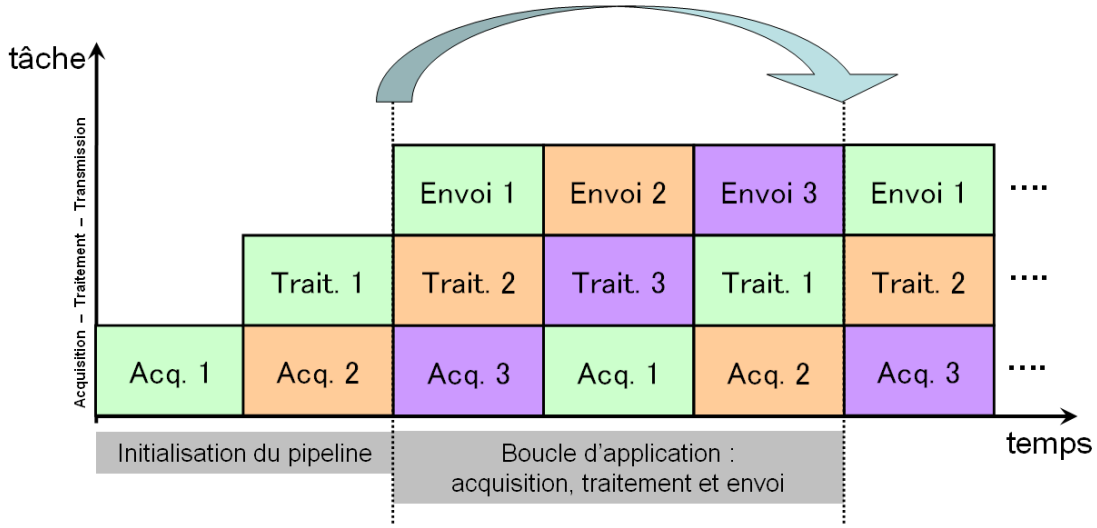


FIG. 5.9 – *Déroulement d'une application acquisition - traitement - envoi, avec utilisation du crossbar pour implémenter une stratégie de "memory swapping" sur les trois blocs mémoire du système.*

5.2.3 Le contrôle des PE's et pilotes

Une des plus grandes difficultés dans la gestion des éléments de la plateforme provient du fait que les différents modules fonctionnent à des cadences qui leur sont propres. Nous défendons que la meilleure façon de synchroniser leur fonctionnement passe par un protocole de contrôle asynchrone, qui permet de se détacher des contraintes et caractéristiques temporelles de chacun d'entre eux. Les modules jouissent donc d'une autonomie de fonctionnement, assurée par un contrôle synchrone local, et sont soumis aux commandes asynchrones provenant de l'unité de contrôle centrale.

Ainsi, le contrôle des éléments périphériques, tels que le capteur d'images et les PE's, est assuré par :

- un ensemble de registres ;
- un ensemble de drapeaux ("flags") permettant la mise en oeuvre du protocole de contrôle ("handshaking" asynchrone).

En fait, du point de vue de la CCU, il n'y a pas de différence entre un pilote matériel et un PE, car les deux sont soumis aux mêmes règles de contrôle, et sont considérés comme des opérateurs. Du point de vue du programmeur, ces éléments sont une partie intégrante du processeur, en étant contrôlés de façon simplifié par un nombre réduit d'instructions, de registres et de flags.

Les registres de sortie sont utilisés pour le passage de paramètres vers les opérateurs. Ils font partie du fichier registre de la CCU, mais sont différents des autres registres dans le sens où leur contenu est disponible en lecture de façon permanente à l'extérieur de l'unité centrale. Par contre, leur chargement est réalisé par exactement les mêmes instructions que pour tous les autres registres.

Tous les opérateurs connectés à un registre de sortie donné ont accès en lecture à son contenu de façon permanente. Ainsi, pour définir par exemple la taille de l'image à acquérir, un registre de sortie peut être chargé avec le nombre de lignes, et un autre avec le nombre de colonnes souhaité. Le pilote du capteur d'images, connecté à ces registres, aura automatiquement accès à ces informations. Dans le cas où un autre opérateur aurait besoin de ces mêmes informations, il suffit de le connecter à ces mêmes registres.

Les registres d'entrée sont analogues aux registres de sortie. Ils sont accessibles en lecture par la CCU, mais ce sont les opérateurs externes qui se chargent de leur écriture. Ils servent à renvoyer des données vers l'unité centrale, pouvant représenter un résultat ou le status d'un opérateur. Bien évidemment, on parle ici de résultat scalaire, comme par exemple la position d'un objet dans l'image calculée par un PE de tracking. Le transfert de données volumineuses est fait à l'aide du crossbar et des blocs mémoire.

Les flags d'entrée et sortie permettent d'implémenter un protocole très simple de "handshaking" asynchrone. Les flags de sortie sont contrôlés par la CCU, et reçus par les opérateurs. De façon analogue, les flags d'entrée sont contrôlés par les opérateurs et reçus par la CCU. Le chronogramme du protocole est illustré dans la figure 5.10.

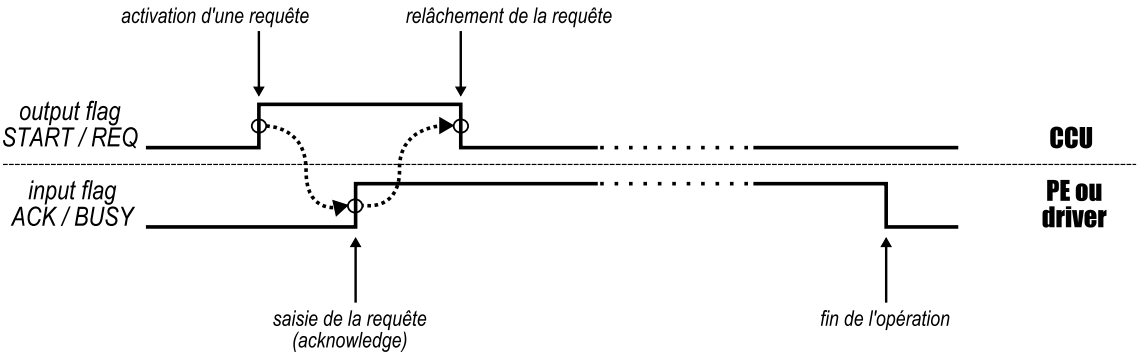


FIG. 5.10 – *Chronogramme du protocole de contrôle des PE's et pilotes hardware.*

Le nombre de flags d'entrée et de sortie disponibles n'est pas fixe, et peut être adapté en fonction de l'architecture implémentée. Ainsi, il est possible d'ajouter facilement une paire de flags afin d'assurer le contrôle d'un nouveau PE, ou d'un nouveau dispositif matériel ajouté à la plateforme.

De façon générale, un flag d'entrée et un flag de sortie suffisent à contrôler le fonctionnement d'un opérateur. La mise à 1 d'un flag de sortie signale à l'opérateur qu'il doit commencer l'opération à laquelle il est destiné (acquisition d'une image, envoi de données vers le système hôte, traitement d'un ensemble de données, etc.). Il répond à cette requête par la mise à 1 d'un flag d'entrée, signalant à la CCU que la requête a été saisie. La CCU peut donc remettre à zéro le flag de sortie, afin de ne pas déclencher une nouvelle opération de façon intempestive, lorsque l'opération en cours arrivera à sa fin. Quand l'opération arrive à sa fin, l'opérateur remet le flag d'entrée qui lui est attribué à zéro, signalant à la CCU que la tâche a été accomplie.

L'utilisation du crossbar en combinaison avec les registres et flags d'entrée/sortie rendent l'inclusion des PE's et pilotes dans l'architecture assez simple. Ceci permet d'une part une flexibilité par rapport à l'application, et d'autre part une flexibilité par rapport à la plateforme.

Du point de vue du programmeur, le **contrôle** de l'application et de la plateforme se retrouvent concentrés dans un seul programme. Les détails concernant le fonctionnement de la caméra lui sont transparents, comme par exemple la génération des signaux au niveau du capteur pour l'acquisition d'une image. Ils seront pris en charge par le pilote matériel dédié, qui assure l'interface entre le composant physique capteur et l'architecture du processeur. Ce pilote est indépendant de l'application, et peut être développé et testé de forme isolé, puis réutilisé à souhait. Les seules règles à respecter sont le protocole de handshaking et la lecture/écriture des données entrantes/sortantes dans une mémoire.

La même chose est valable pour les PE's. Même dans le cas où leur développement reste à la charge du programmeur (pas de librairie, ni IP), ce développement est largement facilité par le découplage du PE par rapport au reste de l'architecture. La programmation indépendante d'un module de traitement, quel que soit le langage utilisé, est bien plus simple que sa conception intégrée à un système. Si les considérations architecturales et de synchronisation peuvent être écartées, le travail de conception se réduit donc à une simple question de programmation fonctionnelle.

5.3 Jeu d'instructions et outil assembleur

Le jeu d'instructions a été conçu dans le but de décrire et de contrôler, de manière simple, l'ordonnancement des tâches qui composent une application, ainsi que les relations entre les éléments de calcul et de contrôle responsables de l'exécution de ces tâches. L'ensemble des instructions décrivant le déroulement de l'application est organisé de façon séquentielle au sein d'un **programme**.

L'outil assembleur permet de transformer les lignes textuelles du programme en code machine binaire, qui sera décodé par la CCU au moment de l'exécution. Chaque instruction du programme en assembleur correspondra alors à une instruction binaire

dans la mémoire programme de la CCU (bijection). Cet outil est écrit en langage C, et supporte certaines fonctionnalités permettant d'alléger la programmation et d'améliorer la lisibilité du programme, comme par exemple l'ajout de commentaires précédés de " // ". Ces commentaires peuvent être placés au début d'une ligne, ou après une instruction, dans la même ligne que celle-ci.

Une autre fonctionnalité permettant d'améliorer la lisibilité est la possibilité d'attribuer des noms aux différents flags, registres et ports du crossbar. Lors de l'assemblage l'outil assembleur remplace ces "étiquettes" par les valeur numériques adéquates. Ainsi, si un registre de sortie est utilisé pour communiquer, par exemple, la taille verticale d'une image (nombre de lignes), il est possible de nommer ce registre SIZE_Y. Ceci augmentera significativement la lisibilité du code, et facilitera la programmation en la rendant plus "parlante". Pour l'instant ces "étiquettes" sont définies au niveau du code C de l'outil assembleur, ce qui ne constitue bien évidemment pas une solution définitive. A terme ces étiquettes devront être définies dans un fichier à part ou dans l'entête même du fichier programme. Ceci permettra d'adapter les nomenclatures en fonction des plateformes et applications, sans modification de l'outil.

D'autres fonctionnalités sont les instructions "raccourci" et l'abstraction du mode d'adressage, qui seront expliquées plus loin dans cette section. Les différentes instructions du jeu d'instructions seront présentées dans la suite. Elles sont organisées par catégorie, selon leur fonction (opération arithmétique, branchements, etc.), à l'exception des instructions NOP et HALT, hors catégorie et présentées ci-dessous :

Mnémonique	**Opérande 1**	**Opérande 2**
NOP		
HALT		

L'instruction NOP ne modifie pas l'état interne du processeur, mis à part le PC, qui est incrémenté. Il s'agit classiquement d'une instruction d'attente pour la synchronisation, mais son usage est limité dû à la présence dans l'architecture de mécanismes de synchronisation plus efficaces (instruction WAIT).

L'instruction HALT arrête l'exécution du programme, laissant le processeur "figé" à son état actuel. Le fonctionnement du processeur n'est repris que lorsqu'un signal de *reset* système est reçu.

5.3.1 Protocole de contrôle des éléments périphériques

Pour contrôler les flags responsables du protocole de handshaking asynchrone, les instructions suivantes sont utilisées :

Mnémonique	Opérande 1	Opérande 2
SET	Flag ID	
WAIT	Flag ID	0 ou 1
RESET	Flag ID	

Les instructions SET et RESET servent à modifier la valeur des flags de sortie. L'instruction WAIT vérifie la valeur des flags d'entrée, et ne permet la continuation de l'exécution du programme que lorsque le flag spécifié prend la valeur indiquée par la deuxième opérande. Ainsi, si par exemple le pilote du capteur d'images est connecté au flag de sortie 3 (START/REQUEST) et au flag d'entrée 2 (ACKNOWLEDGE/BUSY), le lancement d'une acquisition et l'attente de sa fin, selon le protocole indiqué en figure 5.10, sont programmés ainsi :

```
SET Out_F3      //mise à 1 du signal qui déclenche l'acquisition
WAIT In_F2 1    //attente de l'acknowledge de la part du pilote
RESET Out_F3    //remise à zéro de la requête
   ...
   ...          //exécution d'autres tâches
   ...          //indépendantes de l'acquisition en cours
   ...
WAIT In_F2 0    //attente de la fin de l'acquisition
```

Afin de rendre la programmation moins abstraite, il est possible par exemple d'attribuer l'étiquette START_ACQ au flag de sortie 3, et ACQUIRING au flag d'entrée 2. L'étiquetage réduit les erreurs de programmation, permettant de mieux différencier les flags. Le programme devient ainsi :

```
SET START_ACQ       //mise à 1 du signal qui déclenche l'acquisition
WAIT ACQUIRING 1    //attente de l'acknowledge de la part du pilote
RESET START_ACQ     //remise à zéro de la requête
   ...
   ...              //exécution d'autres tâches
   ...              //indépendantes de l'acquisition en cours
   ...
WAIT ACQUIRING 0    //attente de la fin de l'acquisition
```

5.3.2 Le contrôle du *Crossbar*

Afin de gérer l'allocation des mémoires aux différents opérateurs, les instructions suivantes sont utilisées :

Mnémonique	**Opérande 1**	**Opérande 2**
GIVE	Port ID	Mem ID
GET	Mem ID	

L'instruction GIVE alloue la mémoire identifiée par l'opérande 2 à l'opérateur connecté au port identifié par l'opérande 1. L'instruction GET déconnecte une mémoire, afin d'empêcher qu'elle ne soit manipulée de façon intempestive par un opérateur. L'exécution de ces deux instructions consiste dans la modification du mot de contrôle du crossbar (**Crossbar_ctrl**).

Dans l'exemple du tableau ci-dessous, il est démontré comment le mot de contrôle est défini en fonction de la configuration mémoire. Dans cet exemple, nous considérons un crossbar reliant 5 blocs mémoire à 7 ports. Le mot de contrôle est composé de 5 ensembles de 3 bits, un ensemble pour chaque mémoire. La mise à zéro de l'ensemble de bits relatif à une mémoire des-alloue cette mémoire, en coupant sa connexion aux ports d'entrée. La mise d'une valeur différente de zéro connecte cette mémoire au port indiqué par cette valeur. Donc, dans le cas d'un crossbar 5 x 7, le mot **Crossbar_ctrl** est composé de 15 bits, 3 pour chaque bloc mémoire à contrôler. Ainsi, si **Crossbar_ctrl** = 000 000 001 010 011, les mémoires 4 et 5 ne sont pas connectées, la mémoire 1 est connectée au port 3, la 2 au port 2, et la 3 au port 1.

Mémoire	**MEM 5**	**MEM 4**	**MEM 3**	**MEM 2**	**MEM 1**
Allouée au port :	non allouée	non allouée	1	2	3
Crossbar_ctrl	000	000	001	010	011

TAB. 5.2 – *Format du mot de contrôle* ***Crossbar_ctrl*** *pour un crossbar 5 x 7.*

Pour obtenir une telle configuration mémoire, nous devons utiliser les instructions :

```
GIVE Port_3 MEM_1
GIVE Port_2 MEM_2
GIVE Port_1 MEM_3
GET MEM_4
GET MEM_5
```

Si par exemple le pilote du capteur d'images est connecté au port 1 du crossbar, il est possible d'étiqueter ce port IMG_SENSOR. Ainsi,
`GIVE Port_1 MEM_3` laisserai place à
`GIVE IMG_SENSOR MEM_3`, rendant le code plus lisible.

5.3.3 Branchements et appels de routines

Les branchement et sauts sont une catégorie d'instructions de contrôle du programme, permettant de "sauter", par le chargement du compteur programme (PC), vers une adresse quelconque de la mémoire d'instructions.

Les sauts inconditionnels ont lieu systématiquement lors de leur appel. Ils peuvent être utilisés afin de réaliser une boucle de façon infinie (instruction JUMP), ou afin d'appeler une routine ou procédure (instruction CALL). L'instruction RETURN est utilisée pour le retour après exécution de la routine, et permet de reprendre le programme au point où celui-ci s'est arrêté au moment de l'appel. Ceci est possible grâce au stockage, lors de l'exécution de l'instruction CALL, de la valeur du PC dans la pile (Stack). Les instructions JUMP et CALL ont une opérande, contenant l'adresse programme à charger dans le PC. L'instruction RETURN n'a pas d'opérande, car l'adresse à charger est stockée dans la pile.

Mnémonique	Opérande 1	Opérande 2
JUMP	@ Programme	
CALL	@ Programme	
RETURN		

```
0000   JUMP $16   //charge l'adresse 16 dans le PC
...     ...
0016   NOP        //l'instruction JUMP nous mène ici
0017   CALL $32   //charge l'adresse 32 dans le PC, et stocke
                  //dans la pile l'adresse 18 (PC incrémenté)

0018   NOP        //l'instruction RETURN nous mène ici
0019   HALT       //fin de l'exécution
...     ...
0032   NOP        //l'instruction CALL nous mène ici
0033   RETURN     //retour de routine, charge l'adresse 18 dans le PC
```

Les sauts conditionnels sont réalisés ou non, en fonction de la valeur d'un flag. Dans l'architecture implémentée, il existe deux types de sauts conditionnels :

- les sauts conditionnels positifs, réalisés si une condition est vérifiée - instruction BIF (branch if) ;
- les sauts conditionnels négatifs, réalisés si une condition n'est pas vérifiée - instruction BIN (branch if not).

Mnémonique	Opérande 1	Opérande 2
BIF	@ Programme	masque
BIN	@ Programme	masque

Pour ces instructions, l'opérande 1 indique l'adresse de la mémoire programme à atteindre si le saut est effectué. L'opérande 2 contient un masque, qui est comparé aux différents flags du système. La comparaison est faite au moyen d'un `AND` logique appliqué bit à bit entre le masque et le mot formé par la concaténation des différents flags. Si le résultat de la comparaison est différent de zéro, l'instruction BIF réalise le saut, s'il est égal à zéro le saut n'est pas réalisé. Le contraire est valable pour l'instruction BIN.

Le mot qui est comparé au masque est formé par la concaténation des flags de l'ALU, de ceux du comparateur, et des flags d'entrée.

Bit	31..6	5	4	3	2	1	0
	Input_flags(25..0)	`LT`	`GT`	`EQ`	`negative`	`positive`	`zero`

Les flags de l'ALU renvoient le status de la valeur stockée dans l'accumulateur (ACC). Cette valeur est issue de la dernière opération logique ou arithmétique réalisée. Les flags du comparateur sont mis à jour par l'instruction CMP. Cette instruction compare le contenu d'un registre avec le contenu d'un autre registre (Register ID), ou alors avec une valeur définie par adressage immédiat (#valeur), direct ($adresse donnée) ou indirect (*registre d'adresse).

Mnémonique	**Opérande 1**	**Opérande 2**
CMP	Register ID	Register ID, #, $ ou * (valeur)

```
CMP D1 D2   //compare le contenu du registre D1 avec celui
            //du registre D2

CMP D1 #17  //compare le contenu du registre D1 avec la valeur 17

CMP D1 $100 //compare le contenu du registre D1 avec celui stocké
            //dans l'adresse 100 de la mémoire données

CMP D1 *A5  //compare le contenu du registre D1 avec celui stocké
            //dans l'adresse de la mémoire données contenue dans
            //le registre d'adresse A5
```

Les instructions BIF et BIN sont suffisantes pour généraliser un grand nombre de types de sauts conditionnels. L'outil assembleur permet au programmeur d'utiliser un certain nombre de "raccourcis". Ce sont des instructions classiques des langages assembly (e.g. BEQ, BZ, BNE), qui sont reconnues par l'assembleur et traduites vers l'instruction processeur adéquate, par la définition d'un masque approprié. Ces instructions ne comportent que l'opérande 1, qui définie l'adresse cible du saut. Cette opérande reste inchangée lors de la traduction vers l'instruction machine équivalente.

Instruction assembleur	Instruction processeur équivalente	Masque
BZ (Branch if zero)	BIF	0x01h
BNZ (Branch if not zero)	BIN	0x01h
BP (Branch if positive)	BIF	0x02h
BNP (Branch if not positive)	BIN	0x02h
BN (Branch if negative)	BIF	0x04h
BNN (Branch if not negative)	BIN	0x04h
BEQ (Branch if equal)	BIF	0x08h
BNE (Branch if not equal)	BIN	0x08h
BGT (Branch if greater than)	BIF	0x10h
BLT (Branch if less than)	BIF	0x20h

TAB. 5.3 – *Instructions assembleur de saut conditionnel, et leurs instructions processeur et masque équivalents.*

L'avantage de cette stratégie adoptée pour les sauts conditionnels est le fait d'une part de pouvoir réaliser des sauts à conditions composées (branch if negative or equal par exemple), et d'autre part de pouvoir utiliser les flags d'entrée comme condition. Par exemple :

```
CMP ACC #14
BIF $1000 0x0Ch      //masque = 0x0000 1100
BIN $2000 0x11h      //masque = 0x0001 0001
BIF $3000 0x40h      //masque = 0x0100 0000
```

Le saut à l'adresse 1000 est exécuté si la valeur contenue dans accumulateur est négative **ou** égale à 14. Le saut à l'adresse 2000 est exécuté si cette valeur n'est **ni** supérieure à 14, **ni** égale à zéro.

Finalement, le saut à l'adresse 3000 est exécutée si le flag d'entrée 0 est actif, dans le cas où les sauts précédents n'ont pas eu lieu. Il est donc possible d'utiliser les différents flags d'entrée pour changer le déroulement du programme, dans un mécanisme proche de celui des interruptions ou trappes.

5.3.4 Le contrôle et stockage des données

Afin de contrôler le chargement des registres, ainsi que le stockage des variables en mémoire données, les instructions suivantes sont utilisées :

Mnémonique	**Opérande 1**	**Opérande 2**
LOAD	Register ID	#, $ ou * (valeur)
MOV	Register ID	Register ID
STO	$ ou * (@ Donnée)	Register ID ou # (valeur)

Pour les instructions de contrôle et stockage des données l'ordre des opérandes dans l'instruction suit la norme Intel. La première opérande définie la destination de la donnée (registre ou adresse mémoire à écrire), et la deuxième opérande définie sa source (valeur ou adresse mémoire à lire).

L'instruction LOAD permet d'écrire une valeur dans un registre de sortie, d'adresse ou de données. La valeur à écrire peut être définie de trois formes différentes :

- par adressage immédiat - la valeur est donnée directement dans l'instruction. Exemple : `LOAD D4 #17` écrit la valeur numérique 17 dans le registre D4.
- par adressage direct - la valeur est lue dans l'adresse mémoire indiquée dans l'opérande 2. Exemple : `LOAD D3 $150` écrit dans le registre D3 la valeur contenue dans l'adresse 150.
- par adressage indirect - la valeur est lue dans l'adresse mémoire contenue dans un registre d'adresse. Exemple : `LOAD D1 *A3` charge le registre D1 avec la valeur contenue dans l'adresse indiquée par le registre A3.

L'instruction MOV copie la valeur contenue dans un registre vers un autre registre. Ainsi, `MOV Out_R1 In_R0` copie la valeur du registre d'entrée zéro dans le registre de sortie 1.

L'instruction STO écrit une valeur en mémoire. La valeur à écrire peut être le contenu d'un registre, ou une valeur numérique donnée directement dans l'instruction. L'adresse où écrire peut être donnée directement dans l'instruction, ou être contenue dans un registre d'adresse. Voici un exemple de l'utilisation de ces instructions :

```
LOAD D0 #13      //charge le registre D0 avec la valeur 13
                 //D0 <- 13

STO $10 #25      //écrit la valeur 25 à l'adresse 10
                 //Mem[10] <- 25

LOAD A1 $10      //charge le registre A1 avec la valeur
                 //contenue dans l'adresse 10    A1 <- Mem[10]

STO *A1 D0       //écrit le contenu du registre D0 dans l'adresse
                 //contenue dans A1              Mem[A1] <- D0

MOV A1 D0        //copie le contenu de D0 dans A1
                 //A1 <- D0

STO *A1 #40      //écrit la valeur 40 à l'adresse contenue
                 //dans A1                       Mem[A1] <- 40
```

```
LOAD D1 *A1      //charge le registre D1 avec la valeur contenue
                 //dans l'adresse indiquée par A1    D1 <- Mem[A1]

STO $1 D1        //écrit le contenu de D1 à l'adresse 1
                 //Mem[1] <- D1
```

Le lecteur attentif conclura qu'à la fin de l'exécution de ce court programme D0 = 13, D1 = 40 et A1 = 13.

La CCU peut également avoir accès aux autres blocs mémoires de l'architecture, par le biais du bus externe qui est connecté à un port du dispositif crossbar. Les instructions suivantes permettent de réaliser la lecture et l'écriture sur ce bus externe :

Mnémonique	Opérande 1	Opérande 2
EXT_READ	Register ID	$ ou * (@ Donnée)
EXT_WRITE	$ ou * (@ Donnée)	Register ID ou # (valeur)

L'instruction EXT_READ est équivalente à l'instruction LOAD, mais agit sur le bus externe, plutôt que sur la mémoire de données de la CCU. De la même manière, l'instruction EXT_WRITE est équivalente à l'instruction STO. Il est important de souligner que, pour que ces instructions soient effectives, il faut qu'un des blocs mémoire de la plateforme soit alloué au port affecté à la CCU. Dans le cas contraire ces instructions n'auront aucun effet.

```
LOAD A2 #10              //A2 <- 10

GIVE CCU MEM_1           //connecte MEM_1 à la CCU
EXT_READ D0 *A2          //D0 <- MEM_1[A2]
EXT_WRITE *A2 #15        //MEM_1[A2] <- 15
GET MEM_1                //relâche MEM_1

GIVE CCU MEM_2           //connecte MEM_2 à la CCU
EXT_WRITE *A2 D0         //MEM_2[A2] <- D0
EXT_READ D1 $7           //D1 <- MEM_2[7]
EXT_WRITE $7 #33         //MEM_2[7] <- 33
GET MEM_2                //relâche MEM_2

GIVE CCU MEM_3           //connecte MEM_3 à la CCU
EXT_WRITE $7 D1          //MEM_3[7] <- D1
GET MEM_3                //relâche MEM_3
```

Le code ci-dessus aura pour effet de copier le contenu de l'adresse 10 de MEM_1 dans MEM_2, et de le ré-initialiser à 15. Le contenu de l'adresse 7 de MEM_2 et copié dans MEM_3, et puis ré-initialisé à 33.

Du point de vue du décodeur d'instructions, l'opcode d'une instruction LOAD à adressage immédiat et celui d'un LOAD à adressage direct sont différents. L'outil assembleur abstrait le type d'adressage utilisé, et défini l'opcode adéquat lors de l'assemblage du code binaire.

Contrairement aux sauts conditionnels, où plusieurs instructions assembleur sont définies par une même instruction machine (BZ, BEQ et BGT assembleur implémentent un BIF processeur), ici une seule instruction assembleur peut être dérivée en trois instructions machine différentes (LOAD assembleur devient LOAD immédiat, LOAD direct ou LOAD indirect au niveau du processeur).

Ainsi, l'outil assembleur développé ne se résume pas à un traducteur de "strings" vers code binaire. Il dispose également d'un certain nombre de fonctionnalités permettant de simplifier la programmation, comme l'étiquetage des registres et flags, les instructions "raccourci" ou l'abstraction du mode d'adressage. Même s'il ne peut pas pour autant être comparé à un compilateur, car il y a bien une bijection entre l'instruction programmée et une instruction machine, ses fonctionnalités permettent tout de même d'apporter un peu de "haut niveau" à la programmation en assembleur.

5.3.5 Les opérations arithmétiques et logiques

Les opérations sur les données sont effectuées par les instructions suivantes :

Mnémonique	Opérande 1	Opérande 2
ADD	Register ID	Register ID, #, $ ou * (valeur)
SUB	Register ID	Register ID, #, $ ou * (valeur)
MULT	Register ID	Register ID, #, $ ou * (valeur)
SHR	Register ID	Register ID, #, $ ou * (valeur)
SHL	Register ID	Register ID, #, $ ou * (valeur)
OR	Register ID	Register ID, #, $ ou * (valeur)
AND	Register ID	Register ID, #, $ ou * (valeur)
NOT	Register ID	
XOR	Register ID	Register ID, #, $ ou * (valeur)

La première opérande désigne le contenu d'un registre. La deuxième détermine la valeur qui sera mise sur le bus de données. Il peut s'agir du contenu d'un autre registre, d'une valeur immédiate, ou d'une valeur stockée en mémoire données. Dans ce dernier cas, l'adressage peut être fait de façon direct, ou indirect à l'aide d'un registre d'adresse.

L'instruction SUB est réalisée dans l'ordre dans laquelle elle est écrite, c'est à dire :
`SUB D0 D1` aura pour effet `ACC <- (D0 - D1).`

Les instructions SHR (SHift Right) et SHL (SHift Left) réalisent des décalages de bits à droite et à gauche respectivement. L'opérande 2 désigne le nombre de bits à décaler. Normalement il s'agira ici d'une valeur immédiate, entre #1 et #31 pour une CCU 32 bits. Il est important de souligner que les opérations de décalage sont effectuées dans le sens arithmétique du terme. Ainsi, un décalage à gauche produira un zéro à la place du LSB, et un décalage à droite copiera dans le MSB sa valeur d'avant le décalage.

Les instructions OR, AND, NOT et XOR réalisent les opérations logiques correspondantes bit à bit entre les deux opérandes d'entrée, à l'exception de l'instruction NOT qui ne comporte qu'une seule opérande.

L'incrémentation et décrémentation de registres sont des opérations très courantes. Ainsi, l'outil assembleur reconnaît les deux instructions "raccourci" INC et DEC, qui sont traduites au moment de l'assemblage vers des instructions machine ADD et SUB, avec l'opérande 2 fixée à 1.

Instruction assembleur	Instruction processeur équivalente
INC Registre	ADD Registre #1
DEC Registre	SUB Registre #1

TAB. 5.4 – *Instructions "raccourci" d'incrémentation et décrémentation du contenu d'un registre, et leurs équivalents en instruction machine.*

L'ALU ne dispose pas de dispositif de protection ou alerte en relation aux *overflows*. Il appartient au programmeur de contrôler et vérifier la plage de valeur de ses variables afin de les éviter. Néanmoins, ceci ne constitue pas un problème en soi, car la CCU étant essentiellement destinée aux tâches de contrôle, la dynamique offerte par les 32 bits est largement suffisante.

5.3.6 Exemples de programmation

Afin d'illustrer l'utilisation du jeu d'instructions présenté, nous allons décrire d'abord la programmation d'une application simple de traitement de deux listes de données stockées en mémoire, et ensuite la programmation de l'application parallèle d'acquisition, traitement et envoi décrite en section 5.2.2 et illustrée dans la figure 5.9.

5.3.6.1 Application : calcul de produit scalaire

La première application est le calcul du produit scalaire entre deux listes (A et B), de 10 éléments chacune. Il s'agit de calculer la somme accumulée des multiplications élément par élément entre les deux listes. Les listes sont stockées en mémoire données à partir des adresses 1000 (pour A[0]) et 2000 (pour B[0]). Le résultat final (X) est stocké à l'adresse 0. Le calcul réalisé est décrit dans l'équation 5.1.

$$X = \sum_{i=0}^{9} A[i] \times B[i] \tag{5.1}$$

```
            //début du programme
            //registre D1 -> stocke les résultats temporaires, initialisation à 0
000         LOAD D1 #0
            //registre D7 -> compteur de boucle, initialisation à 10
001         LOAD D7 #10
            //A0 et A1 -> pointeurs vers le début des listes A et B respectivement
002         LOAD A0 #1000
003         LOAD A1 #2000

            //début de boucle
            //chargement du registre D0 avec un élément de la liste A (pointé par A0)
004         LOAD D0 *A0
            //multiplication de D0 avec un élément de la liste B (pointé par A1)
005         MULT D0 *A1
            //accumulation et stockage du résultat temporaire
006         ADD ACC D1
007         MOV D1 ACC
            //décrément et stockage du compteur de boucle
008         DEC D7
009         MOV D7 ACC
            //branchement si fin de la boucle (compteur == 0)
010         BZ $16
            //incrémentation et stockage des pointeurs A0 et A1
011         INC A0
012         MOV A0 ACC
013         INC A1
014         MOV A1 ACC
            //retour au début de la boucle
015         JUMP $4

            //sortie de boucle
            //stockage de X, et arrêt du programme
016         STO $0 D1
017         HALT
```

Ce premier exemple contient uniquement de la programmation assez "classique", et le lecteur habitué à la programmation assembleur n'y trouvera aucune nouveauté, mis à part l'illustration du jeu d'instructions présenté précédemment.

5.3.6.2 Parallélisation de l'application acquisition - traitement - envoi

Le deuxième exemple illustre les fonctionnalités de contrôle de l'architecture, qui constituent en fait son atout majeur. Considérons une application de vision embarquée, comprenant une phase d'acquisition d'image, une phase de traitement des données, et une phase d'envoi des données vers le système hôte. Tout d'abord nous décrirons la programmation d'une telle application de façon séquentielle. Ensuite, nous montrerons comment l'application peut être optimisée, afin que les différentes phases soient pipelinées et exécutées de façon concurrente. Pour cela une stratégie de "memory swapping" est utilisée, comme dans l'exemple de la figure 5.9.

Par contre, avant de décrire la programmation de l'application, il est indispensable de faire quelques considérations architecturales sur la plateforme. Nous considérons pour cet exemple une plateforme contenant au moins trois blocs mémoire distincts, reliés au système par le biais du dispositif crossbar. Ces blocs peuvent être alloués tour à tour au pilote du capteur, au PE responsable du traitement ou au pilote de contrôle de l'interface avec le système hôte.

Le pilote du capteur est contrôlé au moyen de :

- 5 registres de sortie, "étiquetés" ADDRESS_X, ADDRESS_Y, SIZE_X, SIZE_Y et AD_BASE_REC ;
- 2 flags de sortie, étiquetés IMG_REC et START_ACQ ;
- 1 flag d'entrée, ACQUIRING ;
- le pilote est connecté au port du crossbar étiqueté IMG_SENSOR.

Les registres ADDRESS_X et ADDRESS_Y configurent l'adresse de l'image à acquérir dans l'espace de la matrice photosensible du capteur (imageur CMOS à adressage aléatoire). SIZE_X et SIZE_Y configurent la taille de l'image, et le flag START_ACQ déclenche l'acquisition. Le flag d'entrée ACQUIRING informe à la CCU que l'acquisition est en cours. Le registre AD_BASE_REC et le flag IMG_REC contrôlent l'enregistrement des images acquises, déterminant l'adresse de base pour l'enregistrement, et l'activation ou non de celui-ci. L'utilité de pouvoir activer ou non l'enregistrement en mémoire d'une image acquise est discutée dans le chapitre suivant.

Le PE est contrôlé au moyen du flag de sortie PE1_READY, qui déclenche son fonctionnement, et du flag d'entrée PE1_BUSY, qui informe à la CCU que le traitement est en cours. Le PE est connecté au port du crossbar étiqueté PE1.

Le pilote de l'interface est contrôlé par les registres AD_BASE_BURST, qui configure l'adresse à partir de laquelle les données doivent être envoyées, et NUMBER_BURST, qui détermine combien de données doivent être transmises. Dans le cas de cette application exemple, on renvoie au système hôte des images entières, de taille VGA, après leur traitement par le PE. Ce pilote est connecté au port

étiqueté BURST. Le flag START_BURST déclenche l'envoi des données, et le flag BURSTING indique que l'envoi est en cours.

Le programme ci-dessous représente la programmation de l'application de façon séquentielle. Dans une première étape les différents registres de contrôle des opérateurs sont configurés. Ensuite est réalisée une boucle constitué d'acquisition, traitement et envoi. Si dans l'exemple l'application est statique (taille et adressage des images configurées une fois pour toutes), il est néanmoins parfaitement envisageable de modifier ces paramètres à chaque itération, par le rechargement des respectifs registres avant le déclenchement de chaque acquisition.

```
          // début du programme
          //chargement des registres d'acquisition (pilote capteur)
000       LOAD ADDRESS_X #0     //adresse horizontale de la fenêtre d'acquisition
001       LOAD ADDRESS_Y #0     //adresse verticale de la fenêtre d'acquisition
002       LOAD SIZE_X #640      //acquisition d'images de taille VGA
003       LOAD SIZE_Y #480
004       LOAD AD_BASE_REC #0   //stocker l'image à partir de l'adresse 0

          //chargement des registres d'envoi  (pilote interface)
005       LOAD AD_BASE_BURST #0      //adresse de base des données à transmettre -> 0
006       LOAD NUMBER_BURST #307200  //nb de données à transmettre = SIZE_X x SIZE_Y

          // début de boucle ***********************************************
          //début de l'acquisition
007       GIVE IMG_SENSOR MEM_1
008       SET IMG_REC
009       SET START_ACQ
010       WAIT ACQUIRING 1
011       RESET START_ACQ

          //attente de la fin de l'acquisition
012       WAIT ACQUIRING 0
013       RESET IMG_REC
014       GET MEM_1

          //début du traitement des données
015       GIVE PE1 MEM_1
016       SET PE1_READY
017       WAIT PE1_BUSY 1
018       RESET PE1_READY

          //attente de la fin du traitement des données en Mémoire 1
019       WAIT PE1_BUSY 0
020       GET MEM_1

          //envoi des données vers le host
021       GIVE BURST MEM_1
022       SET START_BURST
023       WAIT BURSTING 1
024       RESET START_BURST

          //attente de la fin de l'envoi vers le host
025       WAIT BURSTING 0
026       GET MEM_1

          //retour de boucle
027       JUMP $07
```

28 instructions suffisent donc à contrôler l'application dans sa forme la plus simple. Il suffit pour cela que les pilotes et le PE respectent le protocole de la CCU. La seule hypothèse faite sur la nature du PE utilisé est qu'il récupère, en entrée, une image en mémoire, et restitue en résultat une image de mêmes dimensions dans cette même zone mémoire.

Afin d'améliorer la performance temporelle de cette application, il est possible de paralléliser les différentes tâches. La figure 5.11 présente le diagramme d'exécution de la version parallélisée, suivie du code assembleur de son implémentation.

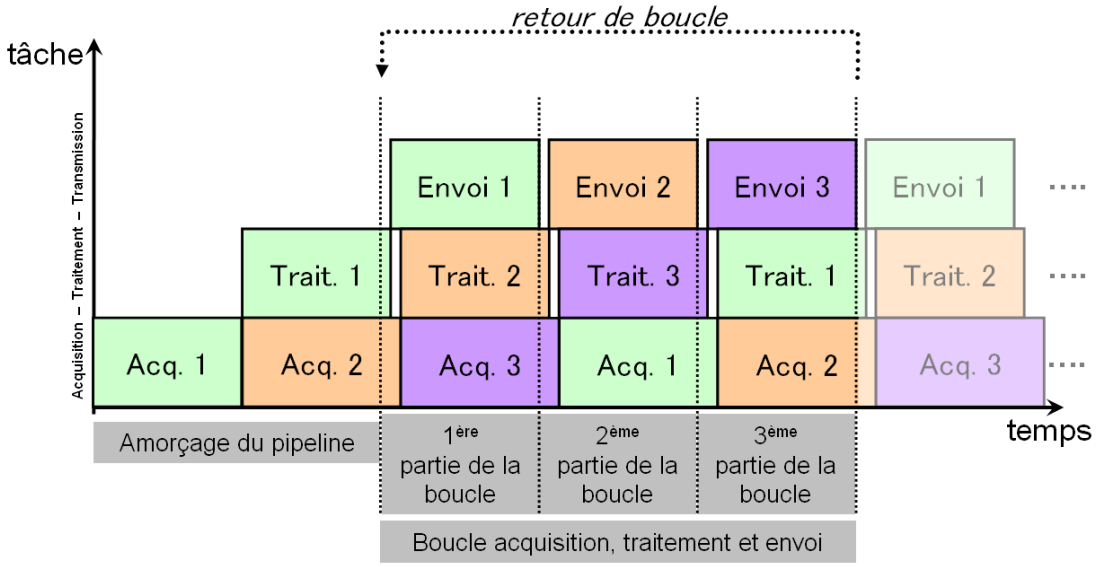


FIG. 5.11 – *Diagramme de l'exécution concurrente des tâches d'acquisition, traitement et envoi, en utilisant une stratégie de memory swapping sur trois blocs mémoire.*

```
            // début du programme
            //chargement des registres d'acquisition (pilote capteur)
000         LOAD ADDRESS_X #0     //adresse horizontale de la fenêtre d'acquisition
001         LOAD ADDRESS_Y #0     //adresse verticale de la fenêtre d'acquisition
002         LOAD SIZE_X #640      //acquisition d'images de taille VGA
003         LOAD SIZE_Y #480
004         LOAD AD_BASE_REC #0   //stocker l'image à partir de l'adresse 0

            //chargement des registres d'envoi  (pilote interface)
005         LOAD AD_BASE_BURST #0      //adresse de base des données à transmettre -> 0
006         LOAD NUMBER_BURST #307200  //nb de données à transmettre = SIZE_X x SIZE_Y

            //************************************************************
            //amorçage du pipeline
            //début de l'acquisition en Mémoire 1
007         GIVE IMG_SENSOR MEM_1
008         SET IMG_REC
009         SET START_ACQ
010         WAIT ACQUIRING 1
011         RESET START_ACQ
```

```
          //attente de la fin de l'acquisition en Mémoire 1
012       WAIT ACQUIRING 0
013       RESET IMG_REC
014       GET MEM_1

          //début du traitement des données en Mémoire 1
015       GIVE PE1 MEM_1
016       SET PE1_READY
017       WAIT PE1_BUSY 1
018       RESET PE1_READY

          //début de l'acquisition en Mémoire 2
019       GIVE IMG_SENSOR MEM_2
020       SET IMG_REC
021       SET START_ACQ
022       WAIT ACQUIRING 1
023       RESET START_ACQ

          //************************************************************
          //début de la boucle --- première partie de la boucle
          //attente de la fin du traitement des données en Mémoire 1
024       WAIT PE1_BUSY 0
025       GET MEM_1

          //envoi des données en Mémoire 1 vers le host
026       GIVE BURST MEM_1
027       SET START_BURST
028       WAIT BURSTING 1
029       RESET START_BURST

          //attente de la fin de l'acquisition en Mémoire 2
030       WAIT ACQUIRING 0
031       RESET IMG_REC
032       GET MEM_2

          //début du traitement des données en Mémoire 2
033       GIVE PE1 MEM_2
034       SET PE1_READY
035       WAIT PE1_BUSY 1
036       RESET PE1_READY

          //début de l'acquisition en Mémoire 3
037       GIVE IMG_SENSOR MEM_3
038       SET IMG_REC
039       SET START_ACQ
040       WAIT ACQUIRING 1
041       RESET START_ACQ

          //attente de la fin de l'envoi des données en Mémoire 1 vers le host
042       WAIT BURSTING 0
043       GET MEM_1

          //************************************************************
          //deuxième partie de la boucle
          //attente de la fin du traitement des données en Mémoire 2
044       WAIT PE1_BUSY 0
045       GET MEM_2

          //envoi des données en Mémoire 2 vers le host
046       GIVE BURST MEM_2
047       SET START_BURST
048       WAIT BURSTING 1
049       RESET START_BURST
```

```
            //attente de la fin de l'acquisition en Mémoire 3
050         WAIT ACQUIRING 0
051         RESET IMG_REC
052         GET MEM_3

            //début du traitement des données en Mémoire 3
053         GIVE PE1 MEM_3
054         SET PE1_READY
055         WAIT PE1_BUSY 1
056         RESET PE1_READY

            //début de l'acquisition en Mémoire 1
057         GIVE IMG_SENSOR MEM_1
058         SET IMG_REC
059         SET START_ACQ
060         WAIT ACQUIRING 1
061         RESET START_ACQ

            //attente de la fin de l'envoi des données en Mémoire 2 vers le host
062         WAIT BURSTING 0
063         GET MEM_2

            //*************************************************************
            //troisième partie de la boucle
            //attente de la fin du traitement des données en Mémoire 3
064         WAIT PE1_BUSY 0
065         GET MEM_3

            //envoi des données en Mémoire 3 vers le host
066         GIVE BURST MEM_3
067         SET START_BURST
068         WAIT BURSTING 1
069         RESET START_BURST

            //attente de la fin de l'acquisition en Mémoire 1
070         WAIT ACQUIRING 0
071         RESET IMG_REC
072         GET MEM_1

            //début du traitement des données en Mémoire 1
073         GIVE PE1 MEM_1
074         SET PE1_READY
075         WAIT PE1_BUSY 1
076         RESET PE1_READY

            //début de l'acquisition en Mémoire 2
077         GIVE IMG_SENSOR MEM_2
078         SET IMG_REC
079         SET START_ACQ
080         WAIT ACQUIRING 1
081         RESET START_ACQ

            //attente de la fin de l'envoi des données en Mémoire 3 vers le host
082         WAIT BURSTING 0
083         GET MEM_3

            //retour à la première partie de la boucle
084         JUMP $24
```

Le code complet compte seulement 85 instructions. Aucune hypothèse n'est faite sur la durée des différentes étapes (acquisition, traitement et envoi). L'utilisation des instructions WAIT suffit pour synchroniser les tâches et respecter leur rapports de dépendance de données, en s'assurant que l'acquisition est finie avant de lancer le traitement, et que le traitement est fini avant de lancer l'envoi. Les particularités du hardware sont transparentes pour le programmeur, qui se préoccupe uniquement de la définition d'un nombre restreint de paramètres pertinents, comme la taille de l'image, et non des dizaines de signaux nécessaires pour contrôler l'acquisition au niveau physique du capteur par exemple.

Le passage entre les différentes mémoires se fait de façon naturelle, sans redirection de pointeur ou indexage d'adresse. Du point de vue du PE, le fait que l'application se fasse en pipeline sur trois mémoires, ou en séquentiel sur une seule n'a aucune importance. Ainsi, le PE peut être développé de façon indépendante, ce qui facilite énormément sa conception.

L'application présentée peut être vue comme un modèle général d'application statique ou passive (acquisition puis traitement puis transmission). Il est possible de changer l'application sans changer le code, en changeant uniquement le module PE connecté. La granularité de ce dernier est variable : il peut s'agir d'un simple calcul de gradient, ou alors d'un détecteur de coins et bords complet.

La granularité du ou des PE's doit être définie par le programmeur : vaut-il mieux avoir un seul PE qui réalise tout le traitement, ou plutôt plusieurs PE's plus simples, responsables chacun d'une tâche précise ? La réponse à cette question dépend fortement des caractéristiques du traitement en question, notamment en ce qui concerne les dépendances de données entre les tâches et les potentielles sources de parallélisme.

Le code présenté ci-dessus peut être également adapté pour décrire une application réactive. Admettons que le PE connecté estime le déplacement entre deux images consécutives (translation horizontale et verticale). Les résultats de ce PE peuvent être renvoyés vers la CCU, par le biais des registres d'entrée, et utilisés pour mettre à jour les registres d'adressage du pilote capteur afin de compenser ce déplacement lors de la prochaine acquisition. L'implémentation d'une application semblable est présentée dans le chapitre 7.

5.4 Processing Elements (PE's)

Dans l'architecture proposée le traitement des données est pris en charge essentiellement par des éléments de traitement dédiés. Ces éléments peuvent être vus comme des accélérateurs matériels, ou des ALU dédiées. Dans le cas de la plateforme SeeMOS par exemple, ces accélérateurs peuvent être implémentés en logique câblée au sein du dispositif FPGA, ou en tant que fonctions C exécutées dans le DSP. En

fait, dans ce dernier cas, c'est le dispositif DSP en lui-même qui est vue comme étant un PE, ou plutôt une "hyper-ALU programmable".

En fait, la notion de PE est très vaste, et dépend fortement de l'application en question, et des choix d'implémentation réalisés par le programmeur (parallèle, séquentielle, granularité des opérateurs, partitionnement FPGA - DSP, etc.). Ainsi, un PE peut contenir d'une simple tâche élémentaire faisant partie d'un algorithme, comme par exemple un calcul de gradient, jusqu'à un ensemble de tâches, composant un traitement complet, encapsulées dans un seul PE (e.g. un PE de suivi de motif, ou de détection de mouvement).

Selon les ressources matérielles de la plateforme cible, les PE's peuvent être implémentés en logique câblée au sein du dispositif FPGA, ou peuvent être constitués de composants externes connectés à ce dernier (DSP notamment).

Parmi les différents types de PE's, nous pouvons considérer essentiellement quatre familles distinctes :

Éléments dédiés : ce sont des éléments qui réalisent une tâche bien précise, et leur flexibilité se résume à la définition d'un petit nombre de paramètres. Leur opération peut être résumée par la lecture en mémoire des données d'entrée, la réalisation d'opérations sur ces données, et l'écriture des résultats.

Éléments génériques ou configurables : plus flexibles que les éléments dédiés, ce sont des éléments conçus pour réaliser une classe de traitements, comme par exemple les opérations de voisinage (corrélation, convolution, etc.). Ils peuvent donc être utilisés pour réaliser différents traitements, selon leur configuration, en analogie à une ALU, qui réalise une opération arithmétique ou logique différente selon l'opcode reçu. Un exemple de ce type d'élément est donné dans la suite de cette section.

Éléments travaillant sur le flot : très semblables aux éléments dédiés, mais contrairement à ceux-ci ils ne travaillent pas sur des données stockées en mémoire, mais directement sur un flot de données provenant d'un capteur ou autre source. Certains traitements, comme par exemple la correction du FPN (Fixed Pattern Noise), se prêtent particulièrement bien à une implémentation "sur le flot".

Éléments programmables : éléments dont l'activation déclenche l'exécution d'une routine logicielle programmée au préalable. C'est le cas notamment du dispositif DSP. L'architecture proposée permet d'utiliser le DSP comme une sorte de "super-ALU" pouvant, selon l'opcode, réaliser des fonctions mathématiques et de traitement des signaux complexes, dont la programmation en VHDL consisterait un véritable défi. Néanmoins, le DSP n'est pas le seul dispositif pouvant s'encadrer dans cette famille. Nous pouvons également imaginer des processeurs SIMD (ASIC externe ou implantation HDL dans le FPGA), voire des micro-processeurs embarquées à part entière (ARM).

Nous présentons ci-suit un modèle de PE câblé configurable, dédié aux opérations de traitement bas-niveau appliquées sur le voisinage d'un pixel. Ce travail a été réalisé dans le cadre de cette thèse et présenté dans [110].

5.4.1 Exemple : un PE configurable dédié aux opérations de voisinage

Les *window-based operations*, ou opérations de voisinage, sont souvent appliquées lors du traitement bas-niveau des images. Filtres gaussiens pour le lissage, opérateurs de corrélation pour l'appariement de motifs, dilatation et érosion morphologiques pour la segmentation, sont autant d'opérations fréquemment utilisées et qui peuvent être qualifiées ou exprimées en tant qu'opérations de voisinage. Un opérateur capable de gérer et exécuter efficacement cette classe d'opérations présente donc un intérêt particulier pour un système de type caméra intelligente.

Afin de concevoir un tel opérateur nous allons d'abord procéder à l'analyse des opérations de voisinage parmi les plus fréquemment utilisées. L'objectif est de démontrer que ces opérations partagent un certain nombre de caractéristiques communes, qui peuvent être extraites afin de concevoir un modèle de calcul général.

L'intérêt d'un tel modèle est de servir comme base pour la création d'un module de calcul générique, pouvant être comparé à une sorte d'ALU spécialisée pour les traitements dits de voisinage. Dans le cadre de la méthode d'implantation proposée dans cette thèse, un tel module peut être envisagé en tant que PE, connecté à l'architecture par le biais du crossbar, et contrôlé par les registres et les flags de la CCU.

Nous considérons qu'une opération de voisinage est appliquée selon un modèle qui reçoit en entrée deux opérandes de type imagette (matrice de pixels ou coefficients, fenêtre ou morceau d'image), et qui résulte dans une valeur scalaire et/ou une nouvelle donnée de type imagette. Ainsi, nous argumentons qu'une opération de ce type est une composition de 3 fonctions, tels que :

$$Q_{(i,j)} = \mathbf{F_M}(\mathbf{F_D}(A_{(i,j)},B_{(i,j)})) \tag{5.2}$$

$$x = \mathbf{F_R}(Q_{(n\times n)}) \tag{5.3}$$

où n est la taille du voisinage, $A_{(n\times n)}$ et $B_{(n\times n)}$ sont les opérandes d'entrée (imagettes), $Q_{(n\times n)}$ est l'imagette résultante et x est le résultat scalaire (fig. 5.12).

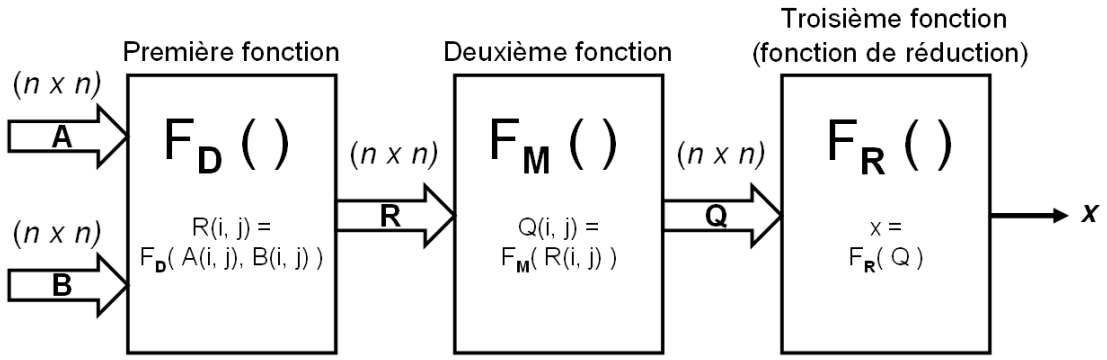


FIG. 5.12 – *Modèle de calcul général pour les opérations de voisinage (*window-based operations*).*

• La première fonction $\mathbf{F_D} : (\mathbb{Z},\mathbb{Z}) \rightarrow \mathbb{Z}$ est appliquée indépendamment pour chaque paire d'éléments des opérandes $A_{(n\times n)}$ et $B_{(n\times n)}$. Le résultat de la fonction $\mathbf{F_D}$ est l'imagette $R_{(n\times n)}$ où :

$$R_{(i,j)} = \mathbf{F_D}(A_{(i,j)},B_{(i,j)}) \tag{5.4}$$

Typiquement, la fonction $\mathbf{F_D}$ est une simple opération logique ou arithmétique.

• La deuxième fonction $\mathbf{F_M} : \mathbb{Z} \rightarrow \mathbb{Z}$ a seulement une opérande de type imagette. Cette fonction est appliquée de façon indépendante pour chaque élément de $R_{(n\times n)}$, résultant dans l'imagette $Q_{(n\times n)}$ telle que :

$$Q_{(i,j)} = \mathbf{F_M}(R_{(i,j)}) \tag{5.5}$$

Classiquement la fonction $\mathbf{F_M}$ est une normalisation, un calcul de valeur absolue, un seuillage, etc...

• La dernière opération est la fonction de réduction $\mathbf{F_R} : (\mathbb{Z}^2) \rightarrow \mathbb{Z}$, appliquée sur les n^2 éléments de l'imagette $Q_{(n\times n)}$, et produisant le résultat scalaire x tel que :

$$x = \mathbf{F_R}(Q_{(n\times n)}) \tag{5.6}$$

Notre objectif est de démontrer que les équations 5.2 et 5.3 peuvent décrire un grand nombre d'opérations de voisinage, selon le choix des fonctions $\mathbf{F_R}$, $\mathbf{F_M}$ et $\mathbf{F_D}$. Pour cela, nous allons analyser quelques exemples parmi les *window-based operations* les plus utilisées.

5.4.1.1 Exemples d'opérations dites de "voisinage"

Convolution :

La convolution est une des opérations de bas-niveau le plus souvent utilisées. Selon le masque de convolution employé, différents résultats peuvent être obtenus (gradient vertical, horizontal, filtrage gaussien, etc.). L'équation 5.7 représente la formule générale de l'opération, où A est une fenêtre de taille $(n \times n)$ autour du pixel (x,y) de l'image d'entrée, et B est le masque de convolution, également de taille $(n \times n)$:

$$\mathbf{Conv}(x,y) = \frac{1}{k} \sum_{(i,j)} \{A(i,j) \times B(i,j)\} \tag{5.7}$$

La constante k est un facteur de normalisation, et dépend de la nature du masque utilisé. L'opération décrite ci-dessus est appliquée pour chaque pixel de l'image d'entrée, et produit un pixel de l'image résultante, excepté pour les zones de bord. Avec quelques arrangements, il est possible d'exprimer le calcul de la convolution autour du pixel (x,y) de la façon suivante :

$$R(i,j) = A(i,j) \times B(i,j) \tag{5.8}$$
$$Q(i,j) = R(i,j)/k \tag{5.9}$$
$$Conv(x,y) = \sum_{(i,j)} Q(i,j) \tag{5.10}$$

Ainsi, les équations 5.2 et 5.3 expriment une opération de convolution quand $\mathbf{F_D}$ est une multiplication, $\mathbf{F_M}$ est une division par k (ou multiplication par $1/k$, ou décalage de bits si k est une puissance de 2), et $\mathbf{F_R}$ est une somme ($\sum$).

Corrélation :

Un autre exemple d'opération de voisinage fréquemment utilisée est le calcul de corrélation. Cette méthode est très courante pour l'appariement de primitives, afin de détecter la présence d'un certain motif (objet ou partie d'objet par exemple) dans une zone de l'image. Le motif recherché est défini par un échantillon d'image de taille $(n \times n)$, qui est ensuite comparé à chaque portion de l'image d'entrée afin de détecter sa présence et position. Une des fonctions de corrélation les plus utilisées est la somme des différences absolues, ou SAD, décrite dans l'équation 5.11, où A est un échantillon d'image autour du pixel (x,y), et B est le motif recherché :

$$\mathbf{SAD}(x,y) = \sum_{(i,j)} |A(i,j) - B(i,j)| \tag{5.11}$$

D'après l'équation 5.11, le calcul de corrélation SAD peut donc être décomposé en trois fonctions: $\mathbf{F_D}$ est une soustraction, $\mathbf{F_M}$ est la fonction valeur absolue, et $\mathbf{F_R}$ est une somme.

Différence d'images :

D'autres exemples de traitements bas-niveau sont la différence d'images (**ImD**, eq. 5.12), et la différence d'images binarisée par seuillage (**b_ImD**, eq. 5.13) :

$$\mathbf{ImD}(i,j) = |A(i,j) - B(i,j)| \tag{5.12}$$

$$\mathbf{b\_ImD}(i,j) = thold\,(A(i,j) - B(i,j)) \tag{5.13}$$

Ces opérations peuvent être appliquées pour la détection de mouvement dans la scène [35]. Les opérandes A et B correspondent à des parties de deux images consécutives d'une séquence. Dans ce cas, la fonction $\mathbf{F_D}$ est une soustraction, $\mathbf{F_M}$ est la fonction valeur absolue pour **ImD** ou la fonction seuillage pour **b_ImD** ($thold()$, eq. 5.14). Comme le résultat de ces opérations est une image, et non pas un scalaire, l'application de la fonction de réduction $\mathbf{F_R}$ n'est pas nécessaire.

La fonction $thold()$ binarise une donnée en fonction de sa valeur et d'un seuil constant et pré-défini :

$$thold\,(X) = (1\ si\ |X| \geq seuil)\,, (0\ sinon) \tag{5.14}$$

Dans un sens strict, la différence d'images n'est pas une opération de voisinage, car le résultat relatif à un pixel ne dépend pas de ses pixels voisins. Néanmoins, le modèle de calcul proposé peut être facilement adapté à cette opération, en supprimant la fonction $\mathbf{F_R}$. Cela veut dire que, si un module de traitement est capable d'implémenter ce modèle de calcul, il sera également capable d'effectuer des opérations de différence d'images, du moment où il soit possible de récupérer en sortie la matrice résultat Q, avant l'application de la fonction de réduction.

Transformations morphologiques :

Des opérations de morphologie mathématique, telles que la dilatation et l'érosion, sont souvent utilisées pour la détection de contours et la segmentation d'images. Ces opérateurs s'encadrent également dans le modèle décrit par une succession de trois fonctions, en considérant l'opérande B comme étant l'*élément structurant* de l'opération.

La décomposition de ces opérations, ainsi que d'autres exemples, sont détaillés dans le tableau 5.5.

Opération	$\mathbf{F_D}$	$\mathbf{F_M}$	$\mathbf{F_R}$
Convolution	$R = A * B$	$Q = R/k$	$x = \sum Q$
Corrélation SAD	$R = A - B$	$Q = \lvert R \rvert$	$x = \sum Q$
Corrélation SSD	$R = A - B$	$Q = R^2$	$x = \sum Q$
Filtre max	$R = A$	$Q = R$	$x = max(Q)$
Différence d'images	$R = A - B$	$Q = \lvert R \rvert$	-
Différence binaire (seuillée)	$R = A - B$	$Q = thold(R)$	-
Différence binaire érodée	$R = A - B$	$Q = thold(R)$	$x = \texttt{AND}(Q)$
Dilatation binaire	$R = A$ `and` B	$Q = R$	$x = \texttt{OR}(Q)$
Érosion binaire	$R = A$ `or` $\overline{B}$	$Q = R$	$x = \texttt{AND}(Q)$
Dilatation en niveaux de gris	$R = A + B$	$Q = R$	$x = max(Q)$
Érosion en niveaux de gris	$R = A - B$	$Q = R$	$x = min(Q)$

TAB. 5.5 – $\mathbf{F_D}$, $\mathbf{F_M}$ *et* $\mathbf{F_R}$ *pour différentes opérations de voisinage.*

Le tableau 5.5 démontre l'adéquation du modèle à trois fonctions des équations 5.2 et 5.3 comme modèle général d'un certain nombre d'opérations de voisinage parmi les plus utilisées. De ce fait, nous proposons la création d'un module de calcul basé sur ce modèle, constituant un PE générique configurable pour l'exécution de ce type d'opérations.

5.4.1.2 Architecture du PE

Les opérations de voisinage requièrent l'exécution d'un nombre élevé d'opérations arithmétiques élémentaires et d'accès mémoire. Si nous considérons par exemple la convolution avec un masque d'une fenêtre de taille $(n \times n)$, seront nécessaires :

- $(2 \times n^2)$ accès mémoire en lecture (chargement de la fenêtre et du masque) ;
- (n^2) multiplications ;
- $(n^2 - 1)$ additions ;
- 1 division ;
- 1 accès mémoire en écriture (stockage du résultat).

Fréquemment, ces opérations seront répétées $(L - n + 1) \times (W - n + 1)$ fois, afin de balayer intégralement une image de taille (L,W). Si aucune optimisation n'est réalisée, le même pixel peut être lu en mémoire jusqu'à n^2 fois. Afin d'atteindre des meilleures performances pour ce type de traitement, deux mesures semblent essentielles : 1) réduire la redondance des accès mémoire et 2) exploiter le parallélisme de données lors des nombreuses opérations répétitives et indépendantes.

L'approche "classique" d'implémentation de ce type de traitement dans les circuits reconfigurables est l'approche *sur le flot*. Le flot de pixels entrants passe par

un pipeline composé de mémoires FIFO intercalées avec des registres. La sortie de ces registres alimente une batterie de multiplieurs et additionneurs qui calculent le résultat. Cette approche est particulièrement intéressante d'un point de vue temporel, car après une latence pour le remplissage du pipeline, un nouveau résultat est calculée à chaque cycle. La cadence de sortie est donc identique à celle d'entrée. Néanmoins cette approche présente deux inconvénients : d'une part elle est gourmande en ressources mémoire internes, particulièrement pour les images de grande taille, dû aux nombreuses FIFO dont la profondeur est liée aux dimensions de l'image. D'autre part, dans le cas d'applications irrégulières où la taille de la fenêtre ou image à traiter est variable, l'adaptation "en ligne" de la profondeur de ces mémoires FIFO peut s'avérer complexe.

Quelques solutions pour pallier ces inconvénients peuvent être trouvées dans [111] et [112]. Nous proposons une solution basée sur le modèle opérationnel discuté précédemment.

Les premières considérations à faire concernent le parallélisme. Comme il est possible de voir dans les équations 5.4 et 5.5, les fonctions $\mathbf{F_D}$ et $\mathbf{F_M}$ sont appliquées indépendamment pour chaque paire de données d'entrée $A_{(i,j)}$ et $B_{(i,j)}$. Cela signifie qu'il y a un parallélisme de données inhérent, avec la possibilité d'appliquer simultanément plusieurs opérations logiques ou arithmétiques dans une structure de type SIMD.

La deuxième source de parallélisme provient de la structure même du modèle de calcul (eqs. 5.2 et 5.3). Les trois fonctions sont exécutées toujours dans le même ordre, et chacune reçoit en entrée les résultats de la fonction précédente. Une structure de type pipeline peut être considérée comme le modèle architectural naturel pour implémenter ce type de processus. Ainsi, afin d'exploiter les deux types de parallélisme possibles (données et tâches), nous proposons un élément de calcul sous la forme d'un pipeline à trois étages, avec les deux premiers étages constitués d'unités SIMD (fig. 5.13).

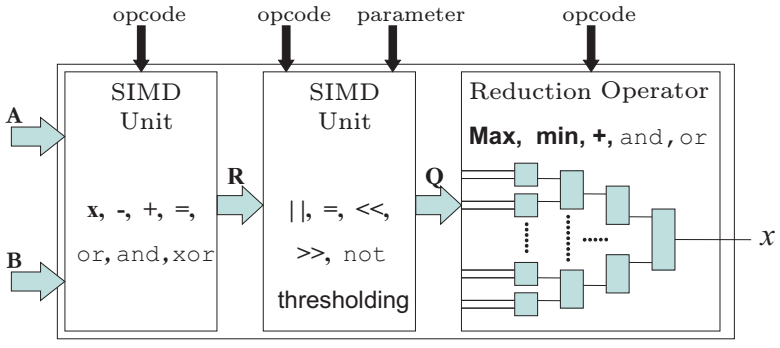


FIG. 5.13 – *Architecture du PE configurable.*

La première unité SIMD implémente la fonction ($\mathbf{F_D}$), étant capable d'exécuter n^2 opérations simultanément, où $(n \times n)$ est la taille définie des opérandes d'entrée (i.e. de la fenêtre d'analyse du voisinage). Les opérations possibles sont l'addition, la soustraction et la multiplication, plus les opérations logiques `AND`, `OR` et `XOR`. Il est également possible d'appliquer une opération d'égalité, c'est à dire de court-circuiter l'opérande d'entrée A vers la sortie. La latence de ce premier étage est d'un cycle d'horloge.

La deuxième unité SIMD implémente la fonction ($\mathbf{F_M}$). Les opérations possibles sont le calcul de valeur absolue, la fonction seuillage, le décalage de bits ou la fonction logique `NOT`. Une entrée est prévue pour la définition d'un paramètre, qui définit la valeur du seuil pour la fonction seuillage, ou le nombre de bits à décaler. La multiplication et la division ne sont prévues dans le jeu d'opérations. Une multiplication est nécessaire pour la corrélation SSD, mais cette fonction est couramment remplacée par la corrélation SAD, qui est plus "légère" d'un point de vue opérationnel, et présente les mêmes propriétés qualitatives que le SSD [113]. Dans les cas où une division est nécessaire, et si le dénominateur n'est pas une puissance de 2, il est plus judicieux de l'effectuer sur le scalaire x après la fonction de réduction, car l'instantiation de plusieurs diviseurs en parallèle est trop coûteuse du point de vue hardware. La latence de ce deuxième étage est d'un cycle d'horloge.

Le troisième étage constitue la fonction de réduction ($\mathbf{F_R}$). Les opérations possibles sont les fonctions logiques `AND` et `OR`, la somme et l'extraction de la valeur maximale ou minimale. Ces opérations sont appliquées sur les données d'entrée en les combinant deux à deux, dans une structure d'arbre dyadique. Pour une matrice d'entrée $(n \times n)$, $2 \times \log_2 n$ étages internes sont nécessaires. Ce nombre d'étages internes détermine la latence pour ce troisième niveau du pipeline en cycles d'horloge. La latence totale du pipeline est donc $2 + 2 \times \log_2 n$.

Pour des opérations telles que la différence d'images, la fonction de réduction n'est pas appliquée. Dans ces cas la matrice Q peut être récupérée directement en sortie, avant le dernier étage du pipeline.

Le chargement des opérandes dans les registres d'entrée de l'élément de calcul peut être pris en charge par un module dédié et autonome, semblable à un contrôleur DMA, ou directement par la CCU en passant par le bus externe. La configuration de l'opération à appliquer à chaque étage, ainsi que le paramètre du deuxième étage sont définis par l'unité de contrôle globale au moyen d'un opcode. Selon les options de contrôle, le PE sera capable d'exécuter différentes opérations, fonctionnant comme une sorte d'ALU dédiée aux opérations de voisinage. Dans le cas où seulement une ou deux opérations seraient nécessaires par étage, une version "allégée" du PE peut être générée afin d'optimiser l'implantation en termes de consommation de ressources hardware.

La stratégie de chargement des registres est d'une importance vitale dans le but de réduire la redondance des accès mémoire. Comme il a été dit précédemment, le même pixel peut être utilisé dans jusqu'à n^2 calculs différents. Une façon simple d'optimiser le chargement des registres est la possibilité de décaler la matrice des registres d'entrée. Si une fenêtre de taille n^2 est chargée pour un calcul, et si le calcul suivant est exécuté sur cette même fenêtre décalée d'un pixel à droite (balayage horizontal), la matrice des registres d'entrée contient déjà $(n-1)n$ valeurs qui peuvent être réutilisées. Dans ce cas, il est possible de décaler la matrice d'une colonne, et charger seulement les n nouveaux éléments, en divisant ainsi par n la redondance des accès mémoire. Ce décalage de la matrice des registres d'entrée est commandée par l'unité responsable de leur chargement.

Si l'opérande B est une constante (ce qui est fréquent), elle sera chargée une seule fois. Si ce n'est pas le cas, une mémoire double port (ou deux blocs mémoire séparés) peut être utilisée pour permettre le chargement simultané des deux matrices de registres.

Une dernière considération concerne le choix de la dimension des opérandes d'entrée (n). Comme la performance de la mémoire a une grande influence sur la performance globale du PE, une grande valeur de n sera inutile si la structure mémoire n'est pas capable de suivre la cadence du pipeline. L'accélération obtenue par la parallélisation des opérations serait compensée par le surcoût dû aux accès mémoire. Le type de la mémoire, sa structure et sa cadence de fonctionnement doivent être soigneusement pris en compte, afin d'assurer que le PE sera capable de délivrer sa performance optimale pour une certaine valeur de n.

Chapitre 6

Implémentation dans la plateforme SeeMOS

Afin de démontrer la validité de l'approche méthodologique présentée dans le chapitre précédent, nous l'avons appliquée à un cas concret, en utilisant comme support d'implémentation la caméra intelligente SeeMOS, décrite en section 3.3.

Pour cela, il a été nécessaire de concevoir et implémenter en VHDL les différentes parties de l'architecture, y compris les pilotes matériels dédiés au contrôle des éléments périphériques de la plateforme, tels que les capteurs et l'interface de communication. Ce chapitre est consacré à la description de l'implémentation de ces différentes parties du système.

6.1 Version de base de l'architecture

L'architecture proposée dans la section 5.2 a été implémentée en langage VHDL. Afin d'évaluer son taux d'occupation spatiale au sein du dispositif FPGA, une version minimale de base a été réalisée.

Cette version minimale comprend les éléments principaux de l'architecture, y compris tous ceux nécessaires au contrôle complet de la plateforme SeeMOS. Ces éléments sont : le coeur de la CCU, un crossbar 5×7 et les pilotes des différents éléments hardware (capteur d'images, capteurs inertiels, DSP et interface Firewire). Ces différents pilotes respectent le protocole de contrôle proposé, en assurant ainsi leur intégration au sein du système et en permettant le contrôle des dispositifs matériels à partir du jeu d'instructions du processeur. Le crossbar et les pilotes hardware utilisés seront présentés plus loin dans ce chapitre.

La version "soft-core" de la CCU a été obtenue au moyen de la retranscription en VHDL du code SpecC de sa version "off-line". Cette dernière ayant été modélisée au niveau RTL, le passage en VHDL a pu être réalisé facilement par la simple adaptation syntaxique de la description en langage SpecC vers une description en VHDL.

On considère l'architecture obtenue comme "minimale" car aucun PE ni fonctionnalité relatifs à une application spécifique n'est implémenté. Seulement les éléments indispensables au contrôle de la plateforme matérielle et à la gestion du système sont présents. Ainsi, la CCU contient uniquement les registres et flags nécessaires au contrôle des pilotes, à savoir :

- 4 flags d'entrée :
  1. INTEGRATING
  2. ACQUIRING
  3. BURSTING
  4. DSP_BUSY

- 6 flags de sortie :
  1. START_INTEG
  2. START_ACQ
  3. IMG_REC
  4. INERT_REC
  5. START_BURST
  6. DSP_READY

- 9 registres de sortie :
  1. INT_TIME
  2. SIZE_X
  3. SIZE_Y
  4. ADDRESS_X
  5. ADDRESS_Y
  6. AD_BASE_REC
  7. AD_BASE_INERT
  8. AD_BASE_BURST
  9. NUMBER_BURST

La fonction et l'utilisation de chacun de ces registres et flags seront détaillées dans les sections suivantes. La figure 6.1 fourni un schéma descriptif de l'architecture implémentée.

D'après la figure, on peut remarquer que 3 ports du crossbar restent disponibles pour l'ajout de PE's. Ils permettront la spécialisation du système vis-à-vis d'une application donnée. Néanmoins, malgré l'absence de PE's spécifiques au traitement, cette architecture minimale constitue tout de même une caméra intelligente à part entière, car la présence du DSP assure sa capacité à réaliser les tâches de traitement d'images.

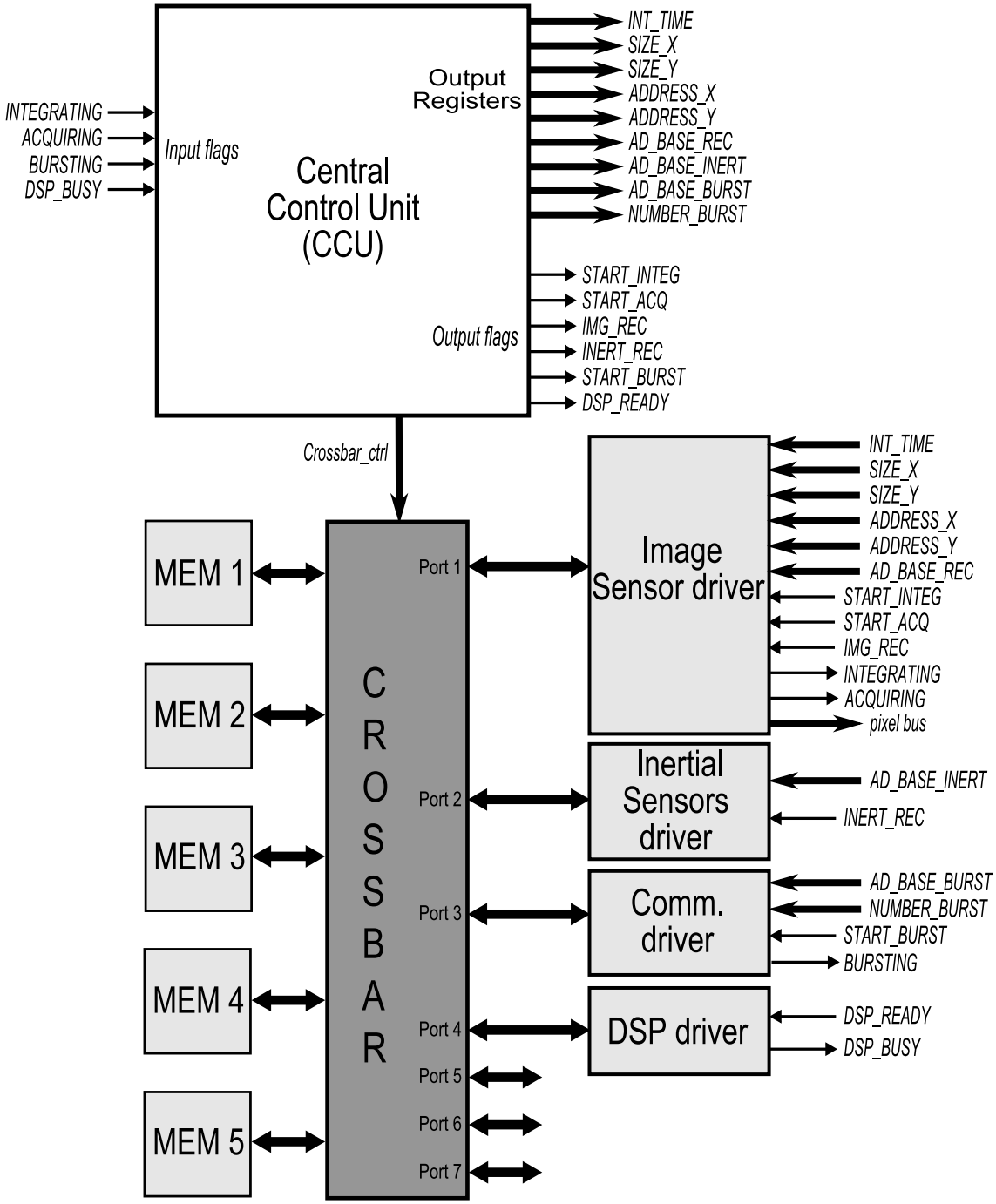


FIG. 6.1 – *Schéma synoptique de l'architecture "minimale" implémentée.*

Le tableau 6.1 détaille l'utilisation des ressources FPGA engendrée par l'instantiation du système. L'occupation spatiale de l'architecture implémentée est très faible (moins de 5% des ressources logiques du FPGA). Ceci veut dire que plus de 95% des ressources restent disponibles pour l'instantiation des différents PE's.

Famille	Altera Stratix
Modèle	EP1S60F1020C7

	Utilisés	**Total**	**Pourcentage**
Éléments logiques	1.661	57.120	3%
I/O Pins	389	782	50%
Bits mémoire interne	73.728	5.215.104	1%
Éléments DSP bloc 9-bit	0	144	0%
PLL	1	12	8%

TAB. 6.1 – *Utilisation des ressources FPGA de l'implémentation minimale de l'architecture.*

La mémoire interne du FPGA est utilisée pour les mémoires programme et données de la CCU. Dans la version minimale implémentée la mémoire programme a été dimensionnée pour contenir jusqu'à 1024 instructions de 9 octets chacune (9 Ko en tout). Ceci représente moins de 2% de la mémoire totale disponible. La capacité de la mémoire programme est paramétrable, et peut s'adapter facilement aux besoins de l'application. Le restant de la mémoire interne non utilisé par la mémoire programme reste disponible, pouvant être utilisé par la mémoire données ou par les PE's. Ces mémoires étant plus rapides que les mémoires externes de la plateforme, leur utilisation par les PE's est intéressante pour les traitements ayant recours à des nombreux accès mémoire.

Comme dernières remarques nous pouvons ajouter que la CCU est cadencée à 50 MHz, et peut réaliser jusqu'à 16,6 MIPS (millions d'instruction par seconde). S'agissant d'une version minimale, le multiplieur de l'ALU n'a pas été implémenté, ce qui explique l'utilisation de 0 blocs DSP du circuit FPGA.

Le très faible taux d'occupation obtenu par cette version minimale démontre que la méthodologie appliquée permet la gestion d'un système hardware complexe à un coût matériel faible. Ceci implique que la grande majorité des ressources du système peuvent être consacrées aux opérations de traitement du signal, la partie gestion du système se faisant discrète. En effet, l'architecture minimale présentée ici peut être vue comme l'élément de base, sur lequel différents éléments viendront s'ajouter afin d'aboutir à un système dédié à une application donnée.

6.1.1 Le Crossbar

La plateforme SeeMOS dispose de 5 blocs mémoire externes indépendants. Ainsi, afin d'adapter l'architecture proposée à cette plateforme, un crossbar 5×7 a été implémenté. Ce dispositif est donc capable de connecter de façon configurable les 5 blocs mémoires à 7 ports différents, qui peuvent être utilisés par les pilotes, les PE's ou la propre CCU. L'implémentation schématique du crossbar est illustrée en figure 6.2.

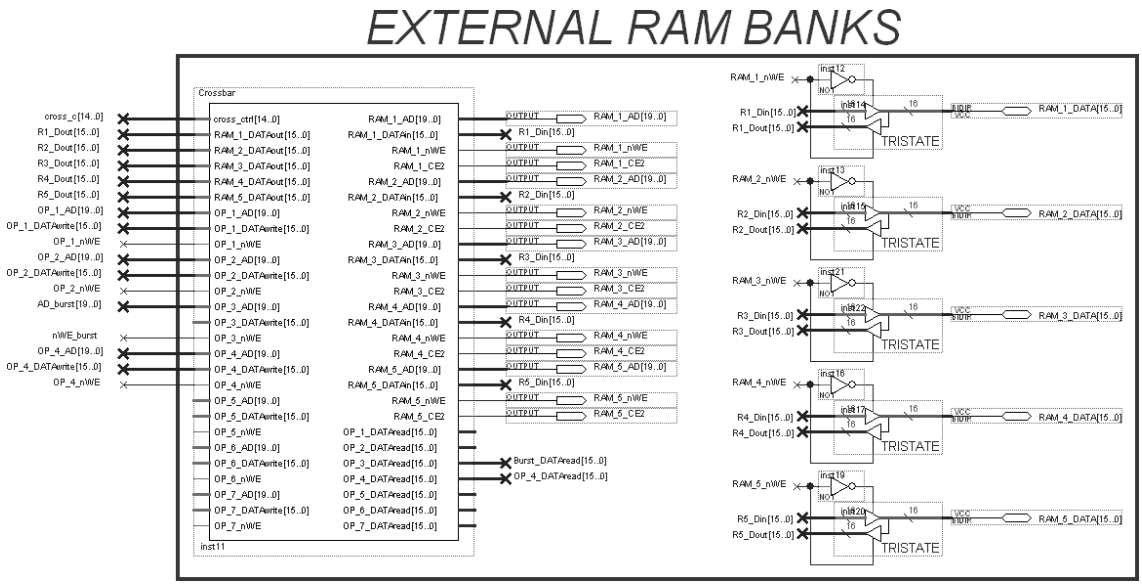


FIG. 6.2 – *Implémentation du crossbar* 5×7 *dans la plateforme SeeMOS.*

La structure crossbar proposée permet une très grande flexibilité dans l'utilisation des ressources mémoire de la plateforme. L'utilisation indépendante et en parallèle des différents blocs par plusieurs PE's ou pilotes, ainsi que la possibilité d'allouer les mémoires de façon dynamique par la simple utilisation des instructions GIVE et GET, permettent la mise en oeuvre de différentes stratégies algorithmiques afin d'optimiser l'implémentation d'une application donnée.

Différentes approches d'implémentation peuvent être facilement réalisées, et même combinées : pipelines, exécution concurrente, memory swapping, etc. Il est important de rappeler que le recours au calcul parallèle ne répond pas uniquement à un besoin d'optimiser les performances temporelles dans l'exécution des tâches. Le parallélisme constitue en effet une caractéristique fondamentale de la vision active et de la vision précoce, qui sont précisément les cadres d'application de la plateforme SeeMOS.

Le crossbar implémenté permet l'écriture simultanée d'une même donnée dans plusieurs blocs mémoires. Si deux blocs ou plus sont alloués à un même port d'entrée,

toutes les opérations d'écriture réalisées sur ce port seront effectuées dans chacune des mémoires allouées. Dans cette configuration, les lectures sur le port en question sont désactivées. Cette fonctionnalité peut s'avérer particulièrement utile dans les cas où deux opérateurs agissent sur un même jeu de données d'entrée, comme par exemple une image. Au lieu d'acquérir cette image dans une mémoire, et puis soit la partager tour à tour entre les deux opérateurs, soit en réaliser une copie, le pilote capteur peut enregistrer l'image acquise sur deux mémoires simultanément, éliminant ainsi tout surcoût lié à la copie ou au partage des données.

Un même opérateur (PE) peut également utiliser plusieurs ports du crossbar. De cette façon, il est possible qu'un PE lise ses données d'entrée dans un bloc, et écrive ses résultats dans un autre.

Enfin, grâce aux différentes possibilités offertes par l'exploitation du crossbar, un très grand nombre d'approches d'implémentation sont accessibles, sans aucune modification nécessaire dans l'architecture du système. Cette possibilité de modifier le chemin de données au sein de l'architecture, sans reprogrammation ou modification matérielle, représente sans nul doute un des points clefs de la méthodologie présentée dans ce manuscrit.

6.2 Le contrôle du capteur d'images

Le capteur LUPA4000 requiert un grand nombre de signaux de contrôle afin de coordonner l'intégration d'une image (accumulation de l'information lumineuse), et son acquisition (lecture des valeurs accumulées). Ces deux étapes peuvent se dérouler simultanément, ce qui constitue un atout majeur permettant d'accroître la cadence d'acquisition du dispositif.

Les nombreux signaux de contrôle sont soumis à des contraintes de durée et synchronisation strictes, ce qui rend difficile le pilotage du capteur par une structure programmable de type *general purpose* (comme un microcontrôleur par exemple). De plus, le contrôle de ces signaux demande une connaissance approfondie du dispositif, ce qui implique une expertise que le programmeur d'applications ne possède pas forcément.

La solution réside en l'utilisation d'un pilote matériel autonome, développé en amont et réutilisable. Ce pilote est soumis à un contrôle "gros grain" de l'unité programmable, qui ne contrôlera en effet qu'un nombre restreint de paramètres essentiels et pertinents du point de vue de l'application, comme par exemple la taille des images ou le temps d'intégration. Le module matériel développé pour le contrôle du capteur d'images est illustré en figure 6.3.

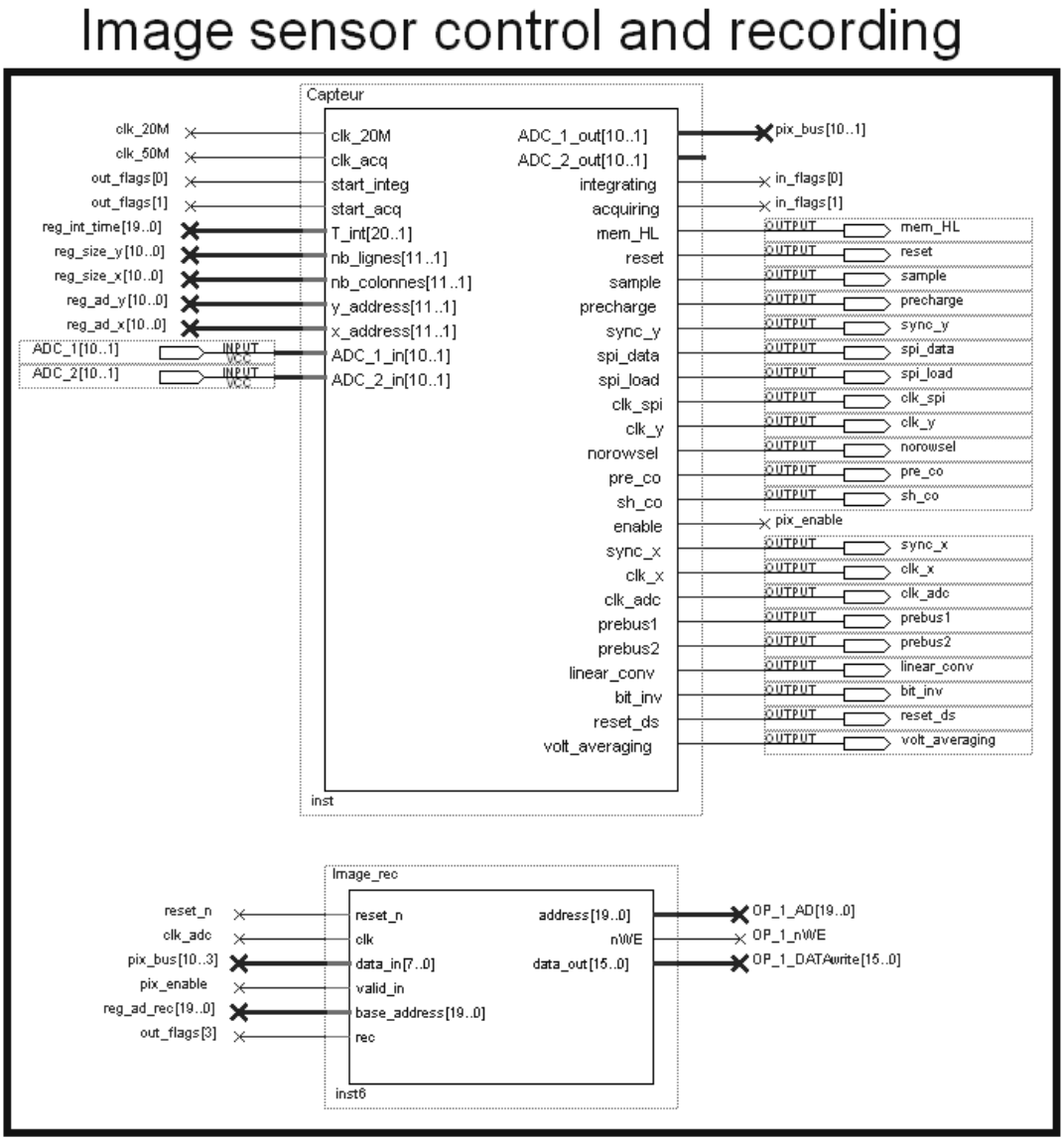


FIG. 6.3 – *Pilote matériel de contrôle du capteur d'images (intégration, acquisition et enregistrement d'images).*

Le pilote développé peut être décomposé en trois parties :

– l'intégration des images ;

– l'acquisition des images ;

– l'enregistrement des images acquises.

L'intégration des images est l'étape qui consiste en l'accumulation, au niveau des photodiodes du capteur, de l'information lumineuse provenant de la scène. Le temps d'intégration nécessaire pour l'obtention d'une image exploitable dépend uniquement de l'éclairage. Dans des conditions normales d'éclairage en intérieur, ce temps est compris entre 5 et $10ms$. Comme dit précédemment, le capteur LUPA4000 permet de réaliser l'acquisition d'une image en même temps que l'intégration de l'image suivante. L'intégration étant un processus relativement long, cette caractéristique du capteur permet d'obtenir des cadences d'acquisition élevées même dans des conditions d'éclairage non-contrôlées. Le pilote développé permet donc de découpler l'acquisition de l'intégration, et ces deux processus sont contrôlés de façon indépendante.

La phase d'intégration n'est caractérisée que par un seul paramètre pertinent du point de vue programmeur : le temps d'intégration. Le contrôle du processus d'intégration est assuré par un registre de sortie (**INT_TIME**), un flag de sortie qui déclenche l'intégration (**START_INTEG**), et un flag d'entrée qui reste à l'état haut pendant que l'intégration est en cours (**INTEGRATING**). La valeur du temps d'intégration est définie en nombre de coups d'horloge à 20Mhz. Ainsi, la valeur 200000 correspond à $10ms$. L'intégration peut donc être contrôlée par l'ensemble d'instructions ci-dessous :

```
//chargement du registre d'intégration
LOAD INT_TIME #200000       //temps d'intégration de 10ms

// début de l'intégration
SET START_INTEG
WAIT INTEGRATING 1
RESET START_INTEG

//attente de la fin de l'intégration
WAIT INTEGRATING 0
```

A la fin du processus d'intégration d'une image, celle-ci est stockée dans une mémoire interne au capteur, et peut finalement être acquise. L'acquisition consiste en la lecture des valeurs des différents pixels dans cette mémoire. Les paramètres pertinents sont la taille de la fenêtre à acquérir (registres **SIZE_X** et **SIZE_Y**), et les coordonnées de son origine (coin supérieur gauche de la fenêtre, registres **ADDRESS_X** et **ADDRESS_Y**). Ces coordonnées sont référencées par rapport au repère capteur, dont l'origine (0,0) est le coin inférieur gauche de la matrice photosensible (cela correspond au coin inférieur droit de la scène observée). Le déclenchement de l'acquisition est fait par l'activation du flag **START_ACQ**. Le flag **ACQUIRING** est ensuite activé par le pilote afin de signaler que l'acquisition est en cours, et reste actif jusqu'à la fin de l'acquisition de la fenêtre spécifiée.

La cadence d'acquisition est de 50 Mpixels par seconde. Les pixels acquis sont mis à disposition dans le bus pixel (signaux `pix_bus[10..1]` et `pix_enable`, figure 6.3).

Une sous-partie du pilote prend en charge le stockage automatique des images dans une mémoire de la plateforme. Afin d'activer le stockage, le flag **IMG_REC** doit être mis à 1. Dans ce cas, les pixels seront écrits un à un dans la mémoire allouée au pilote capteur (en passant par le crossbar), à partir de l'adresse spécifiée par le registre **AD_BASE_REC**. Ce registre pourra être lu par d'autres pilotes ou PE's, afin qu'ils puissent connaître l'adresse où l'image est stockée.

L'utilisation du flag **IMG_REC** peut sembler redondante dans le cas des applications où toutes les images acquises sont stockées directement en mémoire. Par contre, dans certains cas où un PE opère directement sur le flot pixel, il est utile de pouvoir désactiver le stockage automatique. Dans ces situations, le PE en question récupère ses données d'entrée directement sur le bus pixel à la sortie du capteur. Ainsi, le stockage de l'image acquise serait inutile. Pour l'inhiber, il suffit de mettre le flag IMG_REC à 0.

Le programme ci-dessous décrit un processus complet d'intégration, acquisition et enregistrement en mémoire. Afin d'amorcer un pipeline intégration - acquisition, une nouvelle intégration est lancée en parallèle avec l'acquisition de la première image. Ainsi, à la fin de l'exécution de ses instructions, la première image acquise est disponible en mémoire 1, et une nouvelle image est prête pour l'acquisition (stockée dans la mémoire interne du capteur).

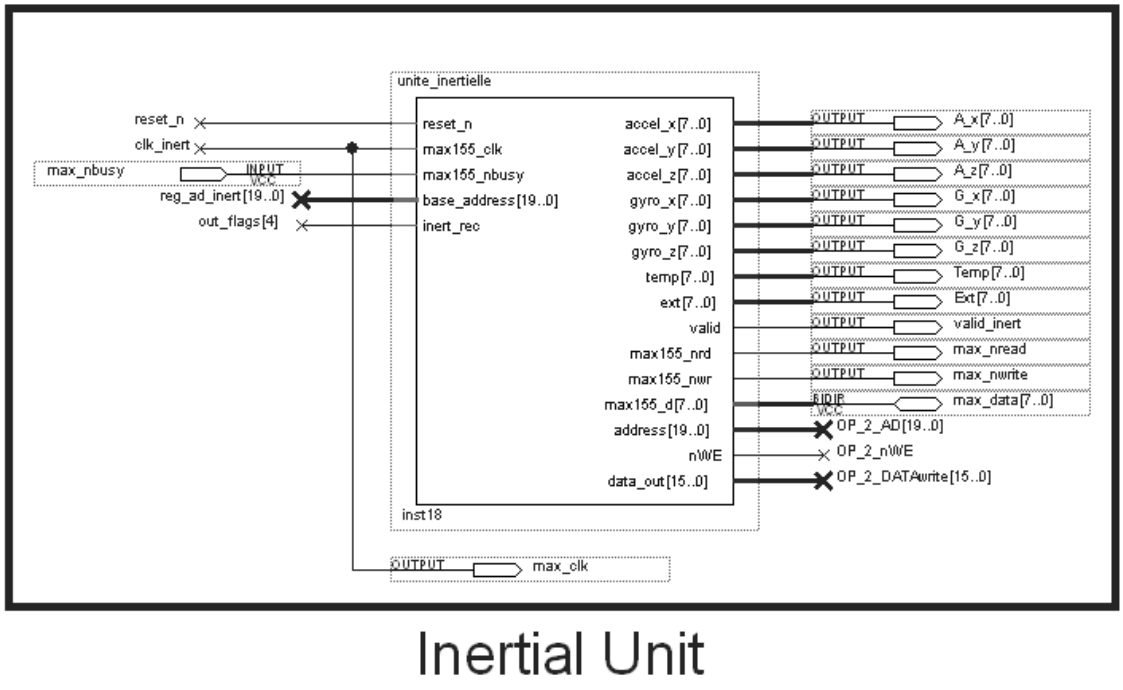
```
//chargement du registre d'intégration
LOAD INT_TIME #200000       //temps d'intégration de 10ms

//chargement des registres d'acquisition
LOAD ADDRESS_X #1536        //adresse horizontale de la fenêtre d'acquisition
LOAD ADDRESS_Y #1536        //adresse verticale de la fenêtre d'acquisition
LOAD SIZE_X #1024           //images de taille 1Mpixel (1024 x 1024)
LOAD SIZE_Y #1024
LOAD AD_BASE_REC #0         //stocker en mémoire à partir de l'adresse 0

// début de la première intégration
SET START_INTEG
WAIT INTEGRATING 1
RESET START_INTEG

//attente de la fin de la première intégration
WAIT INTEGRATING 0

//début de la première acquisition
GIVE IMG_SENSOR MEM_1
SET IMG_REC
SET START_ACQ
WAIT ACQUIRING 1
RESET START_ACQ

//début de la deuxième intégration
SET START_INTEG
WAIT INTEGRATING 1
RESET START_INTEG
```

```
//attente de la fin de la première acquisition
WAIT ACQUIRING 0
RESET IMG_REC
GET MEM_1

//attente de la fin de la deuxième intégration
WAIT INTEGRATING 0

// ...
// ...
// ...
//suite du programme
```

6.3 Le contrôle des capteurs inertiels

Le contrôle des capteurs inertiels est réalisé par la programmation et la lecture du composant ADC, responsable par la numérisation des signaux provenant des accéléromètres et des gyroscopes.

Lorsque l'acquisition des données inertielles est activée, l'ADC est programmé pour procéder à des séquences de lecture des différents capteurs inertiels. Les valeurs lues sont stockées en mémoire, et aussi mises à disposition en tant que signaux de sortie. Ces signaux peuvent, par exemple, être utilisés par la CCU pour adapter l'acquisition des images aux mouvements de la caméra.



Inertial Unit

FIG. 6.4 – *Pilote matériel de contrôle et lecture des capteurs inertiels.*

La figure 6.4 illustre le pilote matériel responsable du contrôle du convertisseur (ADC). Le seul paramètre contrôlé par le programmeur est l'adresse mémoire à partir de laquelle les données acquises sont stockées (registre **AD_BASE_INERT**). Lorsque le flag **INERT_REC** est activé, le module procède tout d'abord à la configuration de l'ADC. Cette étape a pour but de paramétrer le dispositif pour une lecture séquentielle des 6 capteurs inertiels (3 accéléromètres et 3 gyroscopes). Une fois le paramétrage effectué, les mesures sont réalisées à intervalles réguliers. La fréquence des mesures est paramétrable, mais ce paramétrage est réalisé *off-line* au niveau du pilote.

A chaque cycle de mesure, les 6 valeurs lues par l'ADC sont stockées dans la mémoire allouée au module inertiel, en commençant pour le premier cycle par l'adresse de base définie par le registre **AD_BASE_INERT**. L'adresse de stockage est ensuite incrémentée à chaque nouvelle écriture, afin de ne pas réécrire les valeurs précédentes. Les mesures sont effectuées jusqu'à ce que le flag **INERT_REC** soit remis à 0.

A la fin d'un cycle de mesure les nouvelles valeurs sont mises à jour sur les ports de sortie du pilote, qui pourront alors être utilisés directement par la CCU ou par un PE pour obtenir la dernière valeur mesurée, sans obligation de passer par la mémoire.

```
//chargement des registres de la centrale inertielle
LOAD AD_BASE_INERT #0

//début de l'acquisition inertielle
GIVE INERT MEM_3
SET INERT_REC
 ...
 ...
 ...
//arrêt de l'acquisition inertielle
RESET INERT_REC
GET MEM_3
```

Ces quelques instructions ci-dessus sont suffisantes pour contrôler l'acquisition et l'enregistrement des valeur fournies par les capteurs inertiels de la plateforme. Dans cet exemple, les valeurs seront enregistrées à partir de l'adresse 0 de la mémoire 3. Ainsi, à l'aide d'un nombre restreint d'instructions (LOAD, SET, RESET, GIVE, GET), il est possible de faire abstraction de toute la complexité associée à la configuration du dispositif ADC, aux lectures successives des différents capteurs, et à l'enregistrement en mémoire des jeux de données résultants de chaque mesure.

Un exemple d'application réalisant l'acquisition synchronisée d'images et de jeux de données inertielles est donnée dans le chapitre suivant (section 7.2).

6.4 Le contrôle de l'interface 1394

L'interface de communication Firewire de la caméra SeeMOS est constituée d'une carte dédiée connectée à la carte FPGA. Les données sont transmises sur 4 canaux sériels. Ainsi, il est nécessaire de mettre les octets en série avant transmission, et de les reconstituer à partir des bits lors de la réception. Ces tâches, ainsi que la synchronisation et la génération des signaux de contrôle, sont prises en compte par un pilote matériel, illustré en figure 6.5.

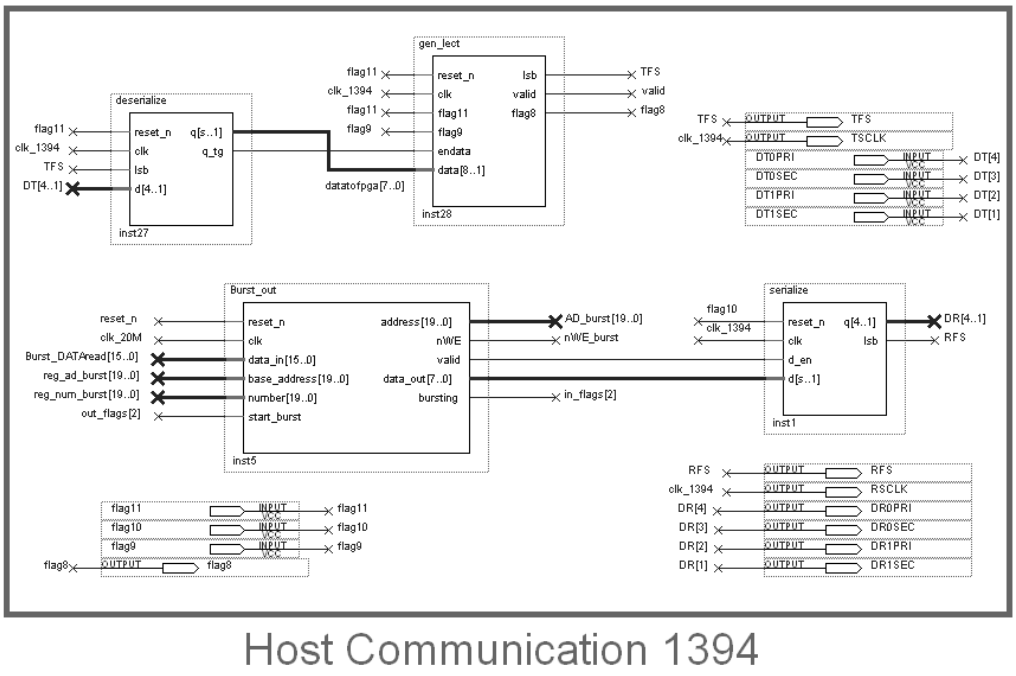


FIG. 6.5 – *Pilote matériel de l'interface de communication avec le système hôte (Firewire - IEEE 1394).*

Du point de vue du programmeur, les détails liés à la carte de communication et au protocole Firewire sont transparents. Pour l'envoi des données vers le système hôte, il suffit de communiquer au pilote l'adresse de base et le nombre de données à transmettre, au moyen de deux registres de sortie de la CCU (**AD_BASE_BURST** et **NUMBER_BURST** respectivement). Un flag vient activer la transmission des données (**START_BURST**), et un autre informe la CCU que la transmission est en cours (**BURSTING**). La fin de la transmission est signalée par la remise du flag **BURSTING** à 0. Bien évidemment, la mémoire contenant les données à transmettre doit être au préalable allouée au pilote. Les instructions dans l'exemple ci-dessous envoient vers l'hôte une image de taille 1000×1000, stockée à l'adresse 2000 de la mémoire 3.

```
//chargement des registres d'envoi  (pilote interface)
LOAD AD_BASE_BURST #2000      //adresse des données à transmettre
LOAD NUMBER_BURST #1000000    //nb de données à transmettre

//envoi des données en Mémoire 3 vers le host
GIVE BURST MEM_3
SET START_BURST
WAIT BURSTING 1
RESET START_BURST

//attente de la fin de l'envoi
WAIT BURSTING 0
GET MEM_3
```

Le gestion des données en provenance du système hôte vers la caméra est prise en compte par un module indépendant. La nature et la fonction des données envoyées vers la caméra étant fortement dépendantes de l'application (certaines applications n'ayant tout simplement pas besoin de renvoyer des données vers la caméra), il paraît judicieux de laisser leur gestion à la charge d'un module spécialisé. Ainsi, le pilote 1394 prend en charge uniquement la remise en forme et la synchronisation des données reçues (désérialisation et activation du signal de validation). Les données reconditionnées sont mises à disposition sur le bus `datatofpga[7..0]`, et sont synchronisées par le signal `valid` (figure 6.5). Le module de gestion approprié viendra se connecter sur ce bus afin de récupérer les données, en fonctionnant de manière autonome et en parallèle avec le reste du système.

Parmi les différentes solutions possibles pour la gestion des données reçues la plus simple est l'écriture dans une mémoire FIFO, qui pourra ensuite être lue par la CCU (au moyen d'une routine logicielle), ou par un autre module câblé.

6.5 Le contrôle du DSP

Le module réalisant l'interface avec le dispositif DSP peut être vu à la fois comme pilote hardware et comme PE.

Pilote hardware parce qu'il traduit les signaux de contrôle de la CCU en signaux de contrôle pour le DSP (génération d'interruptions, etc.), et il adapte le protocole EMIF (External Memory Interface), utilisé par le DSP, à la structure mémoire de la plateforme en le connectant à un port du crossbar.

Par contre, la fonction du DSP étant de traiter des données, il réalise bien une fonction typique de PE. Le DSP est donc vu par le programmeur en tant que PE, mais étant un PE "externe" il nécessite également un module de pilotage matériel.

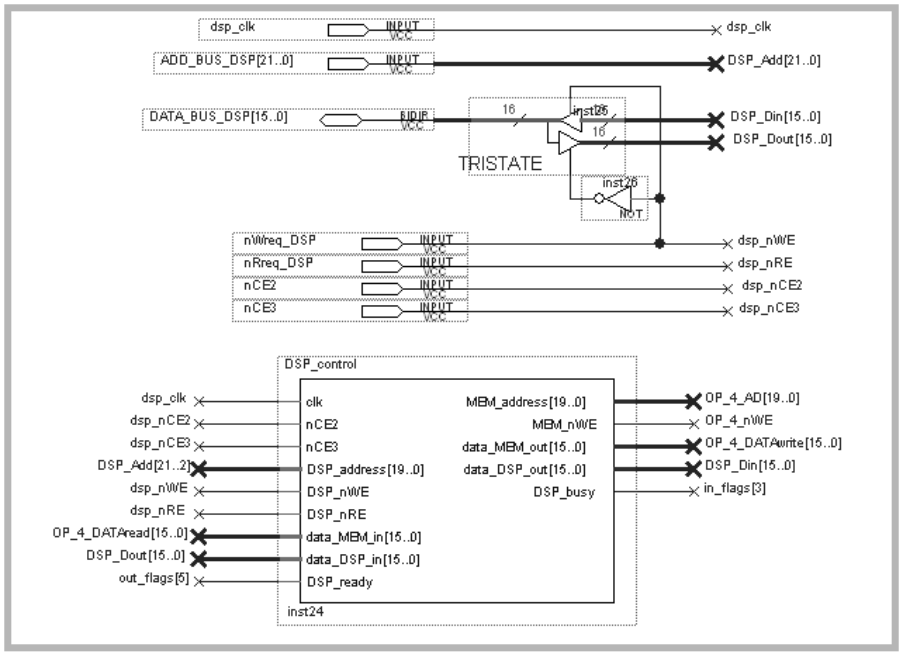


FIG. 6.6 – *Pilote matériel d'interface avec le dispositif DSP (protocole EMIF).*

Ce pilotage sera entièrement pris en charge par ce module et transparent au programmeur, qui viendra contrôler le DSP comme n'importe quel autre PE câblé de son architecture.

La figure 6.6 illustre le module de pilotage développée pour la plateforme SeeMOS. La solution retenue fait appel à l'utilisation de deux "espaces mémoire" distincts. Le premier, contrôlé par le "chip enable" 2 (nCE2) du protocole EMIF, donne accès à un ensemble de registres de lecture ou écriture, qui sont utilisés comme une forme de "tableau noir" pour la communication de paramètres et requêtes entre la CCU et le DSP. Ainsi, la CCU interprète cet espace mémoire comme un ensemble de registres ou flags d'entrée/sortie, tandis que le DSP le voit comme un espace mémoire "traditionnel", accessible par le biais des cycles de lecture ou écriture.

L'utilisation de ce "tableau noir" permet un échange simplifié d'informations entre la CCU et le DSP, et homogénéise le contrôle du DSP par rapport aux autres PE's, d'une part par l'utilisation d'un ensemble de registres et flags d'entrée/sortie, et d'autre part par la connexion à un port du crossbar. Cette connexion avec le crossbar constitue le deuxième espace mémoire utilisé, contrôlé par le "chip enable" 3 (nCE3) du protocole EMIF. Ainsi, le DSP peut accéder directement aux mémoires de la plateforme sans l'intermédiaire de la CCU, en passant par le crossbar en mode DMA. La CCU, et donc le programmeur, s'occupe uniquement de déterminer quel bloc mémoire sera alloué au DSP à un moment donné.

Du point de vue du programmeur, cette méthode rend invisible l'interface entre le composant FPGA et le DSP. Du côté de la CCU, la communication et le contrôle sont réalisés exactement de la même façon qu'avec un PE câblé. Du côté du programme DSP, les registres et flags du "tableau noir", ainsi que la mémoire allouée au moyen du crossbar, sont vus comme étant des variables, au même titre que les variables stockées dans la mémoire propre au DSP. C'est le contrôleur DMA qui se charge de la lecture et écriture des valeurs dans ces mémoires externes, sans ajouter une charge supplémentaire pour le programmeur.

Les quelques instructions ci-dessous illustrent l'appel d'un traitement réalisé par le DSP. Supposons qu'une image soit stockée en mémoire 1. Tout d'abord cette mémoire est allouée au port du crossbar correspondant au DSP (GIVE DSP MEM_1). Ensuite, le flag d'activation du DSP est mis à l'état haut (**DSP_READY**). Du côté du DSP, cette mise à l'état haut sera vue comme le passage à 1 d'une variable booléenne, qui pourra soit déclencher une interruption, soit être utilisée comme condition de sortie d'une boucle d'attente. En conséquence, le DSP signalera la réception de la requête par la mise à 1 d'une autre variable booléenne (**DSP_BUSY**), qui sera écrite dans le tableau noir et qui pourra être lue par la CCU, en tant que flag d'entrée, afin de relâcher la requête. A la fin de l'exécution du traitement, le DSP remet la variable **DSP_BUSY** à 0, permettant à la CCU de récupérer le contrôle sur la mémoire 1 où seront stockés les résultats produits par le traitement.

```
//début du traitement DSP des données en mémoire 1
GIVE DSP MEM_1
SET DSP_READY
WAIT DSP_BUSY 1
RESET DSP_READY

 ...
 ...
 ...
 //attente de la fin du traitement DSP des données en mémoire 1
WAIT DSP_BUSY 0
GET MEM_1
```

Le DSP peut être utilisé comme une ALU programmable, dans le sens où différents types d'opérations peuvent être réalisées au sein d'un même élément. Ces opérations peuvent aller de l'opération arithmétique simple, jusqu'à l'enchaînement de plusieurs routines de traitement plus ou moins complexes. La définition de quel type de traitement doit être exécuté peut être réalisé au moyen d'un opcode, transmis via l'espace mémoire partagé entre le DSP et la CCU (tableau noir). Ainsi, selon de contexte et les besoins de l'application, la CCU peut "commander" au DSP l'exécution d'une routine de traitement ou d'une autre. Bien évidemment, il

est nécessaire que les différentes tâches susceptibles d'être commandées par la CCU soient programmées dans le DSP au-préalable, et associées à un opcode qui permettra au programme DSP d'appeler la bonne routine logicielle lors de l'appel de la CCU.

Les traitements à réaliser dans le DSP sont programmés en langage C. L'accès aux données contenues dans la mémoire externe est fait au moyen d'un pointeur qui adresse l'espace mémoire contrôlé par le chip enable 3 du protocole EMIF. Si l'utilisation de ce pointeur tout au long du traitement est envisageable, les délais relatifs aux accès mémoire externes "individuels" peuvent néanmoins ralentir fortement l'exécution.

Dans les cas où un grand nombre d'accès est nécessaire (par exemple pour la lecture d'une image entière), il est conseillée de d'abord copier l'intégralité des données dans la mémoire du DSP en mode "rafale" ou "burst", via son contrôleur DMA. Une fois les données stockées dans sa mémoire, le DSP pourra réaliser ses calculs de façon plus efficace, et les résultats finaux pourront également être copiés dans la mémoire externe en mode "burst".

Les solutions présentées dans ce chapitre doivent être considérées comme des exemples, et non comme des règles absolues. Il s'agit essentiellement de démontrer comment l'approche méthodologique décrite précédemment a été appliquée à une plateforme donnée, en l'occurrence la plateforme SeeMOS.

Les étiquettes attribuées aux flags et registres ont été définies de façon arbitraire, afin de simplifier la programmation. Celles-ci peuvent être redéfinies librement, par simple modification au niveau de l'outil assembleur.

D'autres solutions sont envisageables, et les pilotes présentés peuvent eux aussi être modifiés ou complétés, afin de supporter des nouvelles fonctionnalités si cela s'avère nécessaire. Néanmoins, la conception ou la modification d'un pilote requiert une bonne connaissance du composant ou du périphérique en question. Ceci n'est pas un point rédhibitoire en soi, car normalement ces pilotes ne sont réécrits que lorsque l'on change de plateforme.

Chapitre 7

Résultats expérimentaux

Dans ce chapitre seront présentés quelques résultats obtenus en suivant la méthodologie proposée pour implémenter des applications sur la plateforme SeeMOS.

Il est important de souligner que l'objectif principal des résultats présentés ici est de démontrer, avant tout, les fonctionnalités de contrôle de la plateforme et de gestion du flot d'application. Cela afin de valider expérimentalement l'approche méthodologique proposée. Les aspects concernant les fonctions de traitement des données ne seront pas pleinement exploités.

En d'autres termes, les applications présentées visent surtout à démontrer comment il est possible de contrôler une structure "hardware" complexe à partir d'un jeu simplifié d'instructions assembleur. Ce sont donc des applications simples du point de vue traitement du signal, mais permettant d'illustrer les différentes possibilités offertes par l'architecture développée et par l'application de l'approche méthodologique proposée dans cette thèse. Les aspects purement mathématiques des algorithmes de vision n'étant pas le thème principal de ce manuscrit, nous ne nous attacherons pas particulièrement sur ce point dans ce chapitre.

7.1 Acquisition en mode caméra rapide

La caractéristique essentielle de toute caméra est bien évidemment sa capacité à acquérir des images. Même si les caméras intelligentes présentent d'autres fonctionnalités, notamment au niveau du traitement embarqué, l'acquisition de séquences d'images reste tout de même un point de passage obligé. L'objectif de ce premier exemple est d'illustrer comment la méthodologie proposée permet, à partir d'un programme simple en langage assembleur, de contrôler l'acquisition d'images de façon efficace.

La possibilité d'agir directement sur les paramètres du capteur, tels que le temps d'intégration et l'adressage des fenêtres, rend possible l'adaptation du processus d'acquisition à différentes situations et contraintes. L'expérience réalisée consiste à observer un phénomène rapide, en se focalisant uniquement sur une petite partie du champ visuel. Pour cela, une séquence d'imagettes (140×140 pixels) a été acquise à une fréquence de 1000 images par seconde.

Une telle cadence d'acquisition est inatteignable avec des caméras "standard", et est propre à des dispositifs coûteux et spécialisés appelés "caméras rapides". Nous ne prétendons pas par cette expérimentation concurrencer ces "caméras rapides", mais plutôt démontrer que l'architecture implémentée permet, avec un faible effort de programmation, d'obtenir des résultats sortant de l'ordinaire par rapport à une caméra classique.

L'expérimentation consiste en filmer la chute de gouttes d'eau dans un verre rempli. Grâce à la haute fréquence d'acquisition il est possible d'observer les différents phénomènes provoqués par l'impact des gouttes sur la surface de l'eau, comme les

reflux et la projection de gouttelettes. Ces phénomènes ne seraient pas observables à une cadence classique d'acquisition de 25 ou 30 images par seconde.

La figure 7.1 illustre plusieurs trames de la séquence. La lecture se fait de gauche à droite et de haut en bas, et l'intervalle entre deux trames consécutives dans la figure est de 15 ms. Ceci veut dire que seulement 1 trame est représentée pour chaque 15 trames acquises.

Afin d'obtenir un *frame rate* de 1000 *fps*, le temps d'intégration a été réglé à 1 ms. Dû au faible temps d'intégration, des conditions spéciales d'éclairage ont été nécessaires, et un éclairage haute fréquence a été utilisé pour illuminer la scène.

La taille des images a été choisie de façon à ce que le temps d'acquisition ou d'envoi d'une image ne dépasse pas 1 ms. L'acquisition étant plus rapide que l'envoi, nous nous sommes basés sur la limite imposée par l'interface de communication (20Moctets/s pour le module 1394 de SeeMOS). Ainsi, chaque image peut contenir un maximum de 20.000 octets, ce qui nous amène aux dimensions de 140×140 pixels. Cette limitation est due au fait que les images sont envoyées en temps-réel vers le système hôte, où elles sont enregistrées.

Le code assembleur utilisé pour programmer la caméra est présenté dans l'annexe B. L'implémentation consiste en un pipeline "intégration - acquisition - envoi", avec *memory swapping* entre acquisition et envoi afin que ces deux tâches puissent se dérouler en parallèle. Le pipeline fonctionnant au rythme de son étape la plus lente (1 ms), la cadence finale obtenue est de 1000 *fps*. L'architecture utilisée est exactement la "version minimale" présentée dans le chapitre précédent (section 6.1).

Le programme en assembleur compte seulement 48 instructions, et il résume tout l'effort de programmation nécessaire pour l'implantation de cette application, car aucune modification n'a été apportée à l'architecture minimale de base. Cela démontre comment la méthodologie d'implémentation proposée dans cette thèse permet de minimiser l'effort d'implantation d'une application, en homogénéisant le contrôle des divers composants matériels de la plateforme derrière un jeu d'instructions unique et dédié.

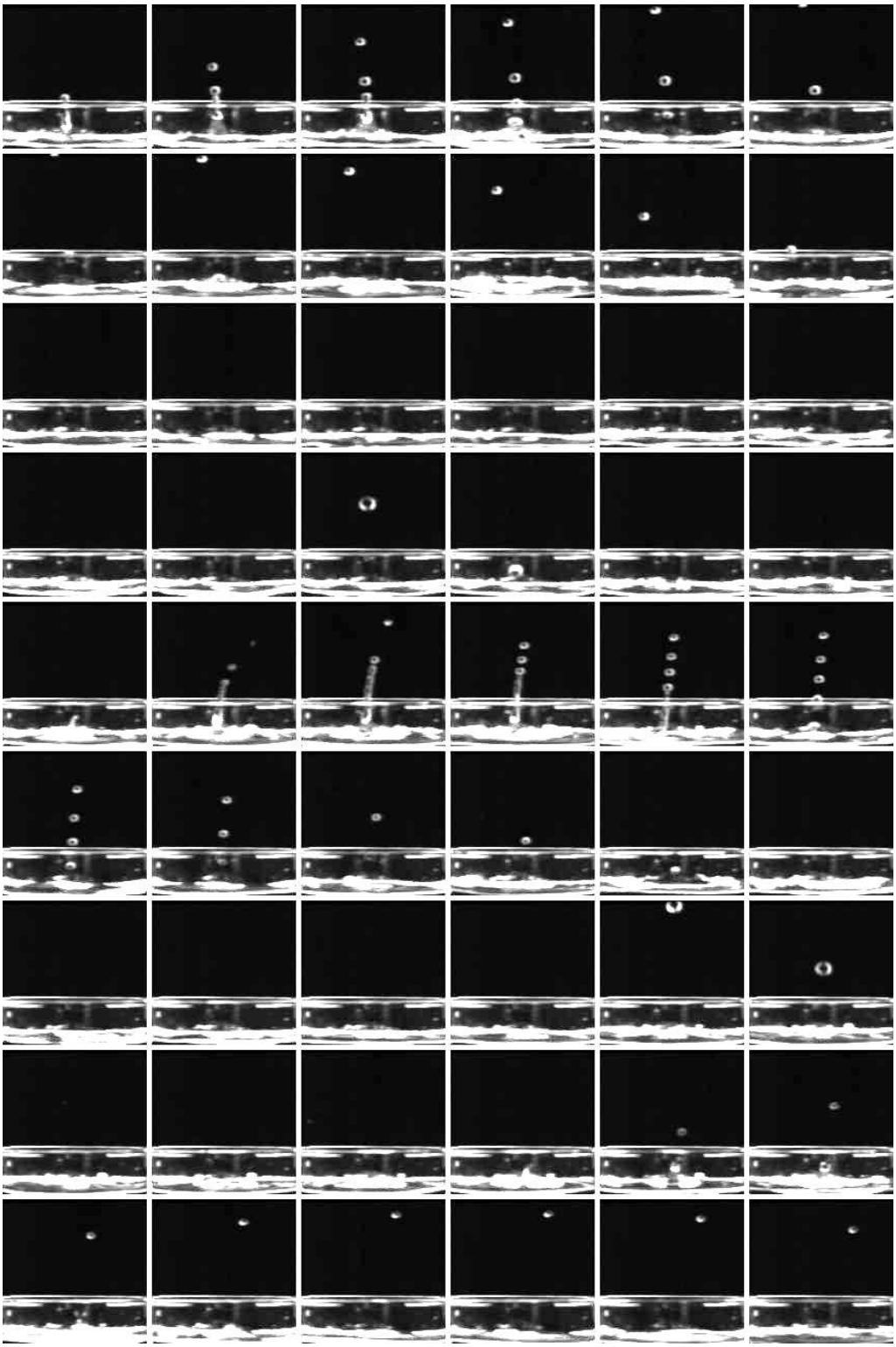


FIG. 7.1 – *Acquisition à 1000 fps de la chute de gouttes d'eau dans un verre rempli, et des projections provoquées.*

7.2 Acquisition synchronisée d'images et de mesures inertielles

Ce deuxième exemple vient compléter l'exemple précèdent en illustrant la gestion de l'aspect multi-capteurs de la plateforme SeeMOS. Équipée d'un ensemble de capteurs inertiels, la caméra SeeMOS est apte à réaliser des acquisitions coordonnées de données visuelles et inertielles. La fusion des mesures visuelles et inertielles permet de relever un certain nombre d'ambiguïtés de la scène, et de remonter à des informations telles que la profondeur et le mouvement propre de la caméra (*egomotion*).

Néanmoins, la gestion des différents capteurs et l'acquisition simultanée et synchronisée des données image et inertielles requièrent un contrôle "fin" des dispositifs de la plateforme. Ceci est vrai notamment au niveau de la structure mémoire, car les différents blocs doivent être successivement re-alloués aux divers opérateurs afin de permettre le déroulement de l'application sans interruption de l'acquisition.

L'expérimentation consiste à acquérir et stocker de façon continue des jeux synchronisés de données image et inertielles, et de les envoyer au système hôte en temps réel. Pendant l'acquisition, la caméra subi un mouvement horizontal latéral d'aller, et puis de retour vers sa position d'origine.

La figure 7.2 présente quelques trames de la séquence acquise, où il est possible de vérifier le mouvement latéral subi par la caméra. La figure 7.3 présente le graphique des d'accélérations mesurées selon l'axe X (axe horizontal, perpendiculaire à l'axe optique). Sur la même figure sont présentés les résultats des deux intégrations successives de ces mesures, afin d'obtenir la vitesse et le déplacement de la caméra. Malgré un signal d'accélération fortement bruité, il est possible de retrouver l'allure du déplacement de la caméra, c'est à dire un mouvement dans une direction, et puis un retour vers sa position d'origine. Les intégrations ont été réalisées *off-line*. Les valeurs numériques dans l'axe des ordonnées n'ont pas de dimension physique directe (pas de calibration des capteurs).

Les images ont une résolution de 2000×500 pixels (affichées partiellement dans la figure 7.2), et sont acquises à une cadence de 20 fps. La fréquence d'échantillonnage des capteurs inertiels a été définie à 2kHz, ce qui représente une centaine de mesures de chacun des 6 capteurs entre deux images successives.

Comme dans l'exemple précédent, l'architecture est implémentée dans sa "version minimale". Le code assembleur de l'application est présenté dans l'annexe C. Seulement 88 instructions sont nécessaires pour programmer une boucle d'acquisition optimisée, prenant en charge la gestion en parallèle des deux modules d'acquisition, des 4 blocs mémoire utilisés, ainsi que de l'interface de communication.

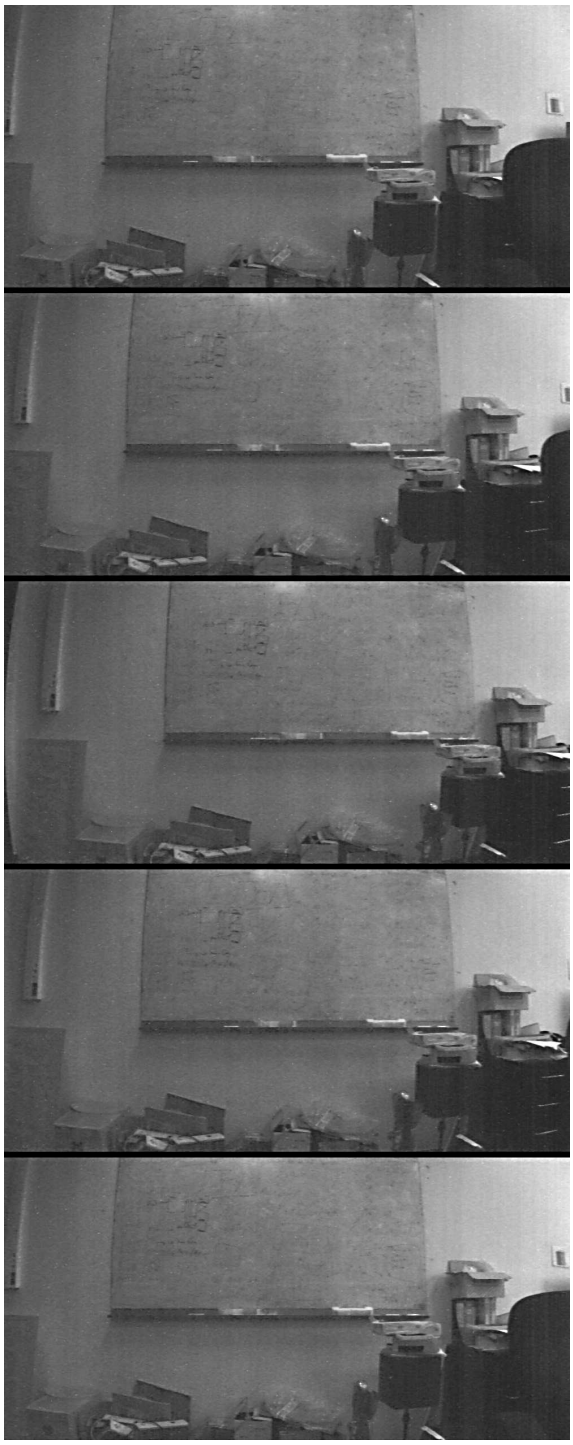


FIG. 7.2 – *Séquence d'images illustrant un mouvement latéral d'*aller-retour *de la caméra.*

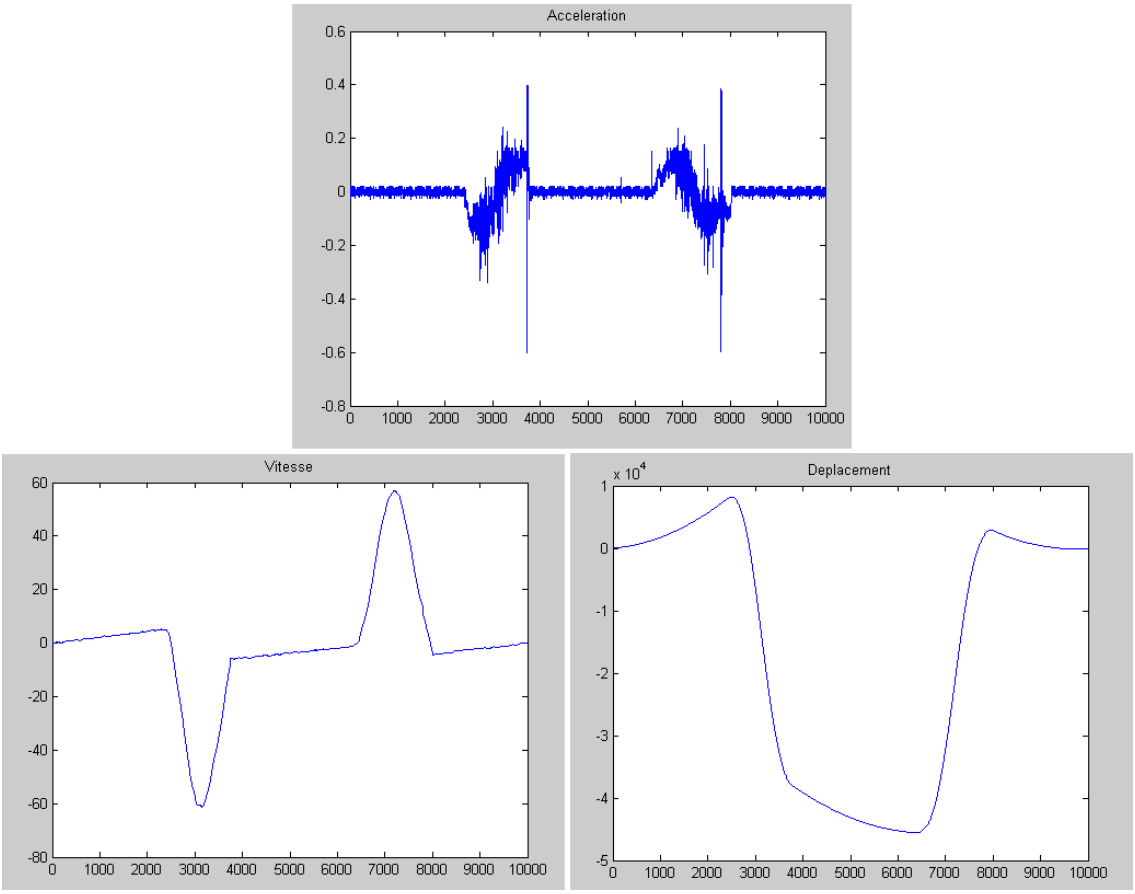


FIG. 7.3 – *Mesure d'accélération en X, et résultats des intégrations successives permettant d'obtenir l'allure de la vitesse et du déplacement de la caméra.*

Malgré son apparente simplicité (acquisition et envoi sans traitement), l'application proposée requiert un contrôle précis des ressources de la plateforme, afin d'assurer un processus d'acquisition synchronisé et ininterrompu. Pour cela, quatre blocs mémoire sont utilisés : deux pour l'acquisition d'images, et deux pour les capteurs inertiels, avec *memory swapping* à chaque itération. Pendant que les acquisitions sont enregistrées dans deux blocs (un image et un inertiel), l'interface de communication transmet le contenu des deux autres vers le système hôte. Ceci permet de tirer profit de façon optimale de la bande passante de communication, tout en assurant une correspondance temporelle précise entre les mesures inertielles et les images acquises.

Un algorithme permettant de retrouver l'estimation de la distance (en Z) d'un objet à partir des données visuo-inertielles de la plateforme SeeMOS a été présenté dans [52].

7.3 Détection de mouvements

L'objectif de cette application est de détecter la présence d'objets en mouvement dans la scène. Dans le plan image, le mouvement des objets se traduit par une variation spatiale et temporelle des niveaux de luminance. L'algorithme utilisé consiste en la détection des variations temporelles par le calcul et l'analyse des différences de niveaux de gris entre deux images consécutives de la séquence. Le résultat est la définition de fenêtres rectangulaires entourant les objets/zones en mouvement dans la scène.

L'algorithme est illustré dans les figures 7.4 et 7.5 [35]. Dans un premier temps, une soustraction pixel à pixel est calculée entre deux trames consécutives. L'image obtenue représente la différence de luminance entre les deux trames en question. Cette image des différences est ensuite seuillée et binarisée, afin d'écarter les petites différences de luminance dues au bruit. Ceci résulte dans une image binaire (au milieu à gauche dans la figure 7.4).

A partir de cette image binaire, la projection verticale est calculée. Cette projection consiste en l'accumulation des valeurs de toutes les lignes, calculée indépendamment pour chaque colonne (en bas à gauche, figure 7.4). Puis, à l'aide d'un détecteur de pics, il est possible d'identifier les bandes verticales susceptibles de contenir des objets mouvants (à gauche dans la figure 7.5).

Finalement, et **uniquement à l'intérieur des bandes verticales détectées précédemment**, la procédure est répétée dans les sens des lignes, en calculant la projection horizontale des zones verticales sélectionnées (au milieu à droite, figure 7.4). Une deuxième application du détecteur de pics permet ainsi de retrouver les frontières verticales de l'objet mouvant (à droite dans la figure 7.5), aboutissant dans la définition d'une zone rectangulaire comprenant celui-ci. Cette méthode permet de détecter et localiser plusieurs zones de mouvement simultanément, comme il peut être vérifié dans la figure 7.4 (en bas à droite). Les deux balles de ping-pong en mouvement ont été correctement détectées et localisées, en temps-réel.

L'implémentation fait appel à un PE câblé, programmé en VHDL et connecté à l'architecture via le crossbar. Ce PE réalise le calcul de la différence absolue entre deux images d'entrée, stockées dans deux mémoires séparées. Puis il réalise l'opération de seuillage afin d'obtenir l'image binaire, qui est finalement écrite dans une troisième mémoire. Le PE est connecté à trois ports du crossbar, étant ainsi capable de lire ses deux opérandes d'entrée et d'écrire un résultat en mémoire simultanément, et à chaque coup d'horloge.

Dans une première version implémentée de cette application l'image binaire des différences est envoyée vers le système hôte. C'est ce dernier qui calcule les projections et applique le détecteur de pics afin de localiser les objets mouvants. Le code assembleur de la CCU utilisé pour cette implémentation est fourni dans l'annexe D.

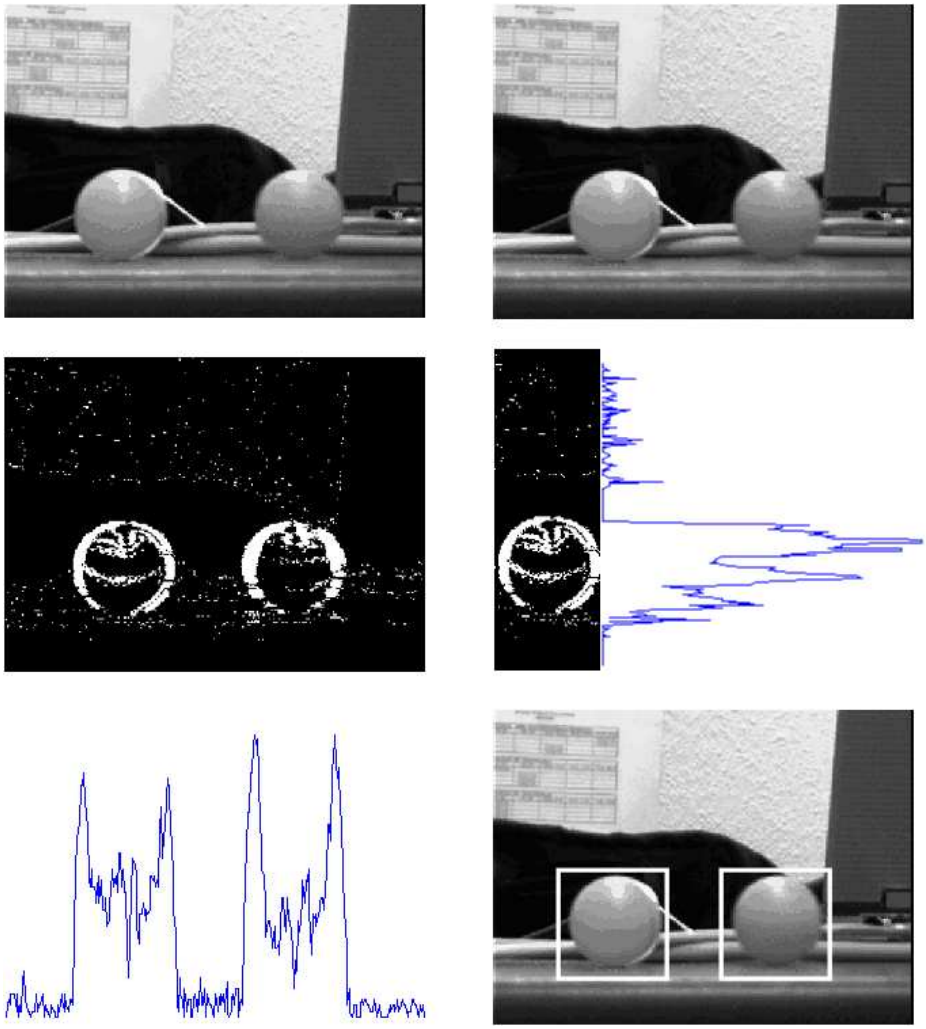


FIG. 7.4 – *Algorithme de détection de mouvements par différence de luminance. En haut : deux trames consécutives. Au milieu à gauche : image des différences seuillée et binarisée. En bas à gauche : projection verticale de l'image binaire (les pics relatifs aux deux balles en mouvement sont bien visibles). Au milieu à droite : première zone verticale d'intérêt, et sa projection horizontale. En bas à droite : résultat final, les deux balles sont correctement détectées et localisées.*

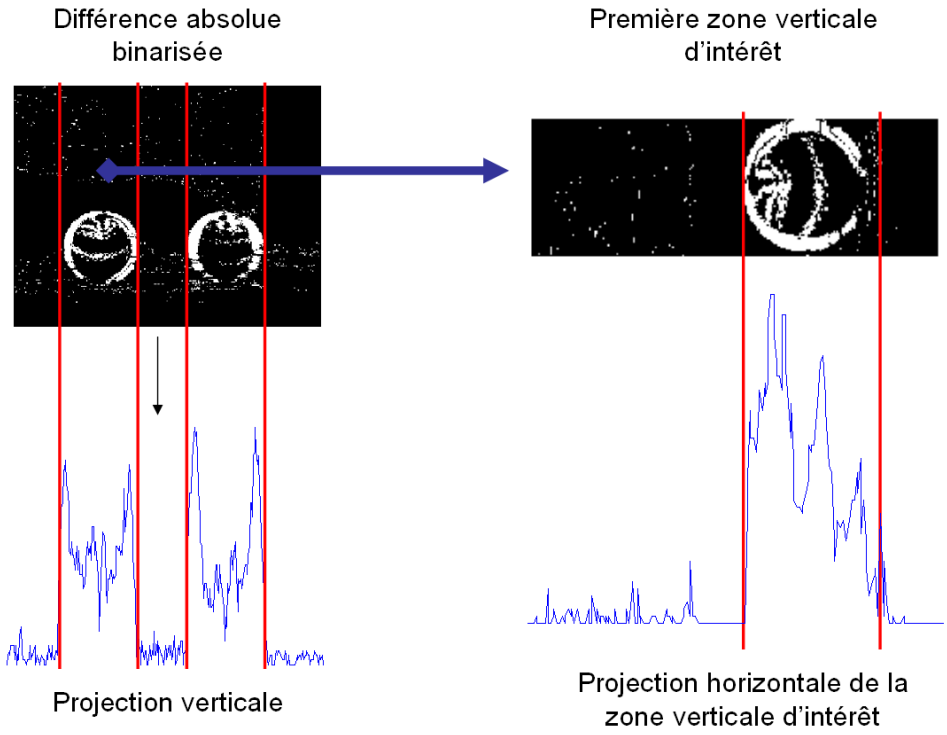


FIG. 7.5 – *Détail de la détection : les lignes verticales indiquent les zones sélectionnées par le détecteur de pics.*

Grâce à la parallélisation du fonctionnement du PE, rendue possible par l'utilisation flexible et parallèle des ressources mémoire de la plateforme, son temps d'exécution est assez bas pour ne pas ralentir le pipeline *"intégration - acquisition - envoi"*. Ainsi, le nouveau pipeline *"intégration - acquisition - traitement - envoi"* fonctionne à la même fréquence que dans le cas où aucun traitement n'est effectué, ayant juste un temps de latence supérieur. La fréquence de fonctionnement est d'environ 64 *fps* pour des images de taille VGA (640×480), ce qui correspond à l'exploitation intégrale de la bande passante de communication entre la caméra et le système hôte. C'est donc la bande passante qui limite la cadence, et non les performances de calcul du système.

Le contrôle du PE est fait au moyen des flags OPER_1_READY et OPER_1_BUSY. Les trois ports utilisés du crossbar ont été étiquetés OP1, OP2 et OP3. Les ports OP1 et OP2 sont les deux ports où le PE lira ses données d'entrée, et OP3 le port où il écrira les données résultantes. Les instructions ci-dessous permettent de contrôler le fonctionnement du PE. Dans cet exemple, les deux images consécutives sont stockées dans les mémoires 1 et 2, et l'image binaire des différences résultante sera écrite en mémoire 4.

```
//début du traitement des données 1 & 2 -> 4
GIVE OP1 MEM_1
GIVE OP2 MEM_2
GIVE OP3 MEM_4
SET OPER_1_READY
WAIT OPER_1_BUSY 1
RESET OPER_1_READY

// ...
// ...
// ...

//attente de la fin du traitement des données 1 & 2 -> 4
WAIT OPER_1_BUSY 0
GET MEM_1
GET MEM_2
GET MEM_4
```

L'optimisation de l'application fait que le pipeline implémenté utilise simultanément les 5 blocs mémoire de la plateforme (trois pour le PE, un pour l'acquisition d'images et un pour l'interface de communication). Ceci engendre une stratégie particulière de "memory swapping", car l'acquisition s'alterne sur trois mémoires (1, 2 et 3), et l'envoi des données sur deux (4 et 5). Afin d'optimiser l'exécution, la boucle d'instructions du pipeline a été déroulée, en explicitant les 6 combinaisons mémoire possibles entre acquisition, traitement et communication.

Par exemple : pendant que le PE opère sur les mémoires 1, 2 et 4, une nouvelle image est acquise en mémoire 3, et les résultats précédents, stockés en mémoire 5, sont envoyés au système hôte. Lors de la prochaine itération le PE utilisera en entrée les images dans les mémoires 2 et 3, et écrira ses résultats en mémoire 5. En même temps, une nouvelle image sera acquise en mémoire 1, et les résultats de l'itération précédente, stockés en mémoire 4, seront envoyés au système hôte.

Une variation de cette application est proposée dans [114]. Celle-ci intègre l'analyse de l'image binaire dans le FPGA, afin de détecter la présence de mouvements dans la scène. Ensuite, et seulement dans les cas où un mouvement est détecté, le DSP est utilisé pour appliquer une égalisation d'histogramme sur les images concernées, afin d'ajuster leur contraste avant l'envoi au système hôte.

7.4 Suivi rapide de motifs par corrélation

L'objectif de cet exemple n'est pas de proposer une méthode robuste ou innovante de tracking, mais plutôt d'illustrer comment le contrôle du capteur peut être mis au profit de ce type d'application. Ceci constitue en effet un des principes fondamentaux de la vision active.

L'algorithme implémenté est basé sur une méthode classique de suivi par appariement de motifs, en utilisant la corrélation SAD (*Sum of the Absolute Differences*). L'appariement de motifs est une opération fréquemment employée pour des applications telles que la stabilisation d'images, ou la construction de mosaïques à partir de plusieurs images (*mosaicing*).

Le fonctionnement de l'algorithme est illustré en figure 7.6 [115] : tout d'abord, le motif à suivre, de taille $(t \times t)$, est défini et enregistré. Puis, avant chaque nouvelle acquisition, une fenêtre de recherche de taille $((t+2\delta) \times (t+2\delta))$ est définie autour de la position estimée du motif recherché (sa position actuelle ou une nouvelle position estimée à partir d'un modèle d'évolution). Le paramètre δ correspond à l'erreur maximale supportée, en pixels, entre l'estimation de position du motif et sa position réelle.

Une fois la fenêtre de recherche définie, elle est acquise et stockée en mémoire. L'opération de corrélation est ensuite appliquée pour chaque zone de taille $(t \times t)$ comprise dans la fenêtre (eq. 7.1). La zone présentant le meilleur score de corrélation (plus petite valeur de SAD) est considérée comme étant la nouvelle position du motif dans l'image. Enfin, la position de la fenêtre de recherche est mise à jour, en compensant l'erreur d'estimation (vecteur $\overrightarrow{E}_k$, eq. 7.2), et en réestimant sa position pour la prochaine trame.

Dans les équations ci-dessous A_k est la fenêtre de recherche à l'itération k, B est le motif suivi, (X_k,Y_k) sont les coordonnées du motif dans la fenêtre de recherche (après appariement), et le vecteur $\overrightarrow{E}_k$ représente l'erreur entre la position estimée et la position mesurée du motif à l'itération k.

$$\begin{aligned} \mathbf{SAD}_k(x,y) &= \sum_{(i,j)=0}^{t-1} |A_k(x+i,y+j) - B(i,j)| \\ avec \quad & 0 \leq x \leq 2\delta \;\; ; \;\; 0 \leq y \leq 2\delta \end{aligned} \tag{7.1}$$

$$\begin{aligned} (X_k,Y_k) &= Argmin(\mathbf{SAD}_k(x,y)) \\ \overrightarrow{E}_k &= (X_k - \delta, Y_k - \delta) \end{aligned} \tag{7.2}$$

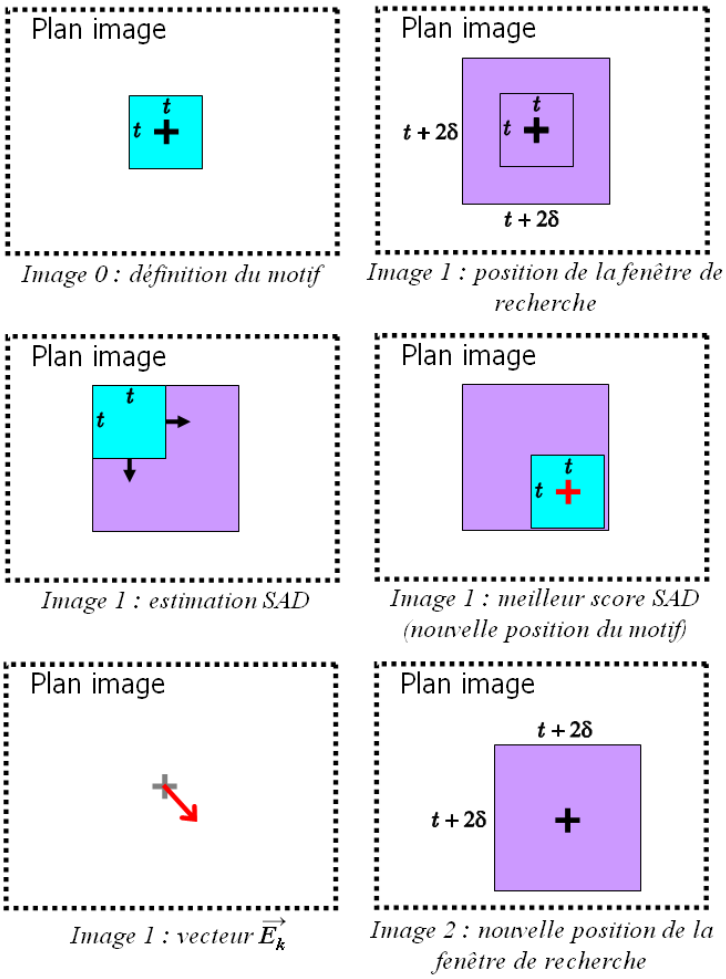


FIG. 7.6 – *Algorithme de suivi par corrélation SAD. Ils ne sont acquis que les pixels appartenant à la fenêtre de recherche.*

L'originalité de l'implémentation proposée consiste dans le fait que seuls les pixels appartenant à la fenêtre de recherche sont effectivement acquis. Grâce à l'adressage aléatoire du capteur CMOS il est possible d'acquérir uniquement la zone de l'image qui sera utilisé pour les calculs de corrélation, permettant un gain substantiel de temps par rapport à une acquisition en pleine résolution. Ce gain de temps permet une fréquence d'acquisition élevée, ce qui est une condition fondamentale pour le bon fonctionnement de l'algorithme de tracking. Ceci s'explique par le fait que plus la cadence de traitement est élevée, plus les déplacements de l'objet suivi entre deux trames sont petits, et plus il y a de chances pour que l'erreur d'estimation soit inférieure à l'erreur maximale supportée δ.

La stratégie consistant à acquérir uniquement les pixels nécessaires au traitement est facilitée par l'architecture implantée. La reconfiguration des paramètres

d'acquisition peut être réalisée à chaque itération, par le simple chargement des deux registres d'adressage capteur de la CCU.

Le calcul d'appariement est réalisé par un PE câblé dédié, responsable de l'exécution des opérations décrites dans les équations 7.1 et 7.2. La corrélation étant un traitement très coûteux en termes d'accès mémoire, les blocs RAM internes du FPGA ont été utilisés, plutôt que les RAM externes (les mémoires internes pouvant fonctionner jusqu'à 200Mhz, contre 83Mhz pour les RAM externes). Ainsi, le PE dispose de deux blocs mémoire dédiés, un pour le stockage du motif à suivre, et un autre pour la fenêtre de recherche. Disposant de ses propres ressources mémoire, le PE n'est pas connecté au crossbar, mais directement au flot pixel en sortie du pilote capteur. L'architecture interne du PE est détaillée dans la figure 7.7.

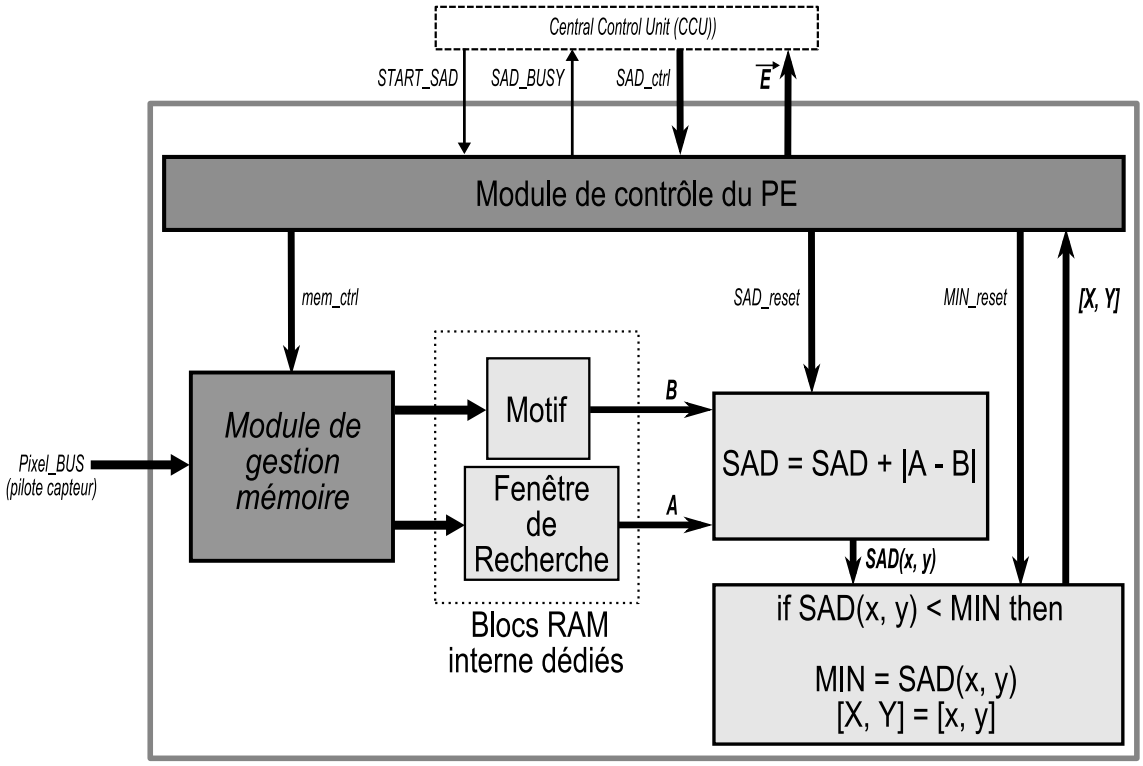


FIG. 7.7 – *Architecture interne du PE dédié d'appariement de motifs par corrélation SAD.*

Commandé par la CCU, le PE enregistre dans un premier temps le motif sélectionné dans une de ses mémoires. Ensuite, à chaque nouvelle itération, il enregistre la fenêtre de recherche dans une deuxième mémoire. Une fois l'acquisition terminée le PE lance le calcul des corrélations sur toute la fenêtre de recherche. A chaque nouvelle valeur de SAD le minima est retenu. A la fin des calculs le vecteur $\overrightarrow{E}_k$ est calculé, et communiqué à la CCU au moyen des registres d'entrée de celle-ci.

La CCU utilise ces informations pour mettre à jour la position de la fenêtre de recherche, avant de lancer une nouvelle acquisition de celle-ci.

Cette mise à jour est réalisée par l'application d'un modèle d'évolution, permettant d'estimer la position du motif dans la prochaine trame à partir des positions actuelle et précédentes mesurées. Des exemples de modèles d'évolution sont le modèle à vitesse nulle (la prochaine position est égale à la position actuelle), et le modèle à accélération nulle (la prochaine position est égale à la position actuelle plus le déplacement observé entre les deux dernières trames).

Les fenêtres acquises sont envoyées au système hôte pour affichage uniquement. Tous les calculs, ainsi que le contrôle de l'application, sont embarqués dans la plateforme caméra intelligente, qui fonctionne en totale autonomie.

L'algorithme, l'implémentation et l'optimisation de cette application de tracking embarqué ont été présentés dans [36] et [115]. Les figures 7.8 et 7.9 illustrent respectivement la sélection du motif à suivre et plusieurs trames résultant du suivi. Les paramètres expérimentaux sont indiqués dans le tableau ci-dessous :

Paramètre	Valeur	Unité
Taille du motif suivi (t)	32 x 32	pixels
δ	9	pixels
Taille de la fenêtre de recherche	50 x 50	pixels
Frame rate	55,6	fps

TAB. 7.1 – *Paramètres expérimentaux de l'application de tracking.*

L'objet auquel appartient le motif sélectionné réalise des mouvements aléatoires sur un plan parallèle au plan optique, y compris des petites rotations et changements de distance (changement d'échelle au niveau image). L'illumination de la scène est hétérogène, comportant aussi bien des zones sombres que des zones saturées de lumière.

Les résultats de la figure 7.9 démontrent que la méthode de suivi implémentée est capable de suivre correctement l'objet, avec une fréquence de traitement relativement élevée (plus de 50 fps), et en supportant ainsi des mouvements rapides aussi bien de la part de l'objet cible que de la caméra elle-même. Il est possible de suivre l'objet sur toute l'étendue du champ visuel de la caméra (2048×2048 pixels). Même si la méthode de suivi utilisée n'assure pas la robustesse aux rotations et changements d'échelle et luminosité, le système est toutefois capable de les tolérer jusqu'à une certaine mesure. L'explication provient du fait qu'une fréquence d'acquisition élevée permet l'utilisation de fenêtres de recherche réduites, limitant ainsi la probabilité de faux appariement du motif recherché. Néanmoins, cette tolérance est purement empirique, et dépend fortement de la nature du motif (symétrie radiale, contraste, etc.).

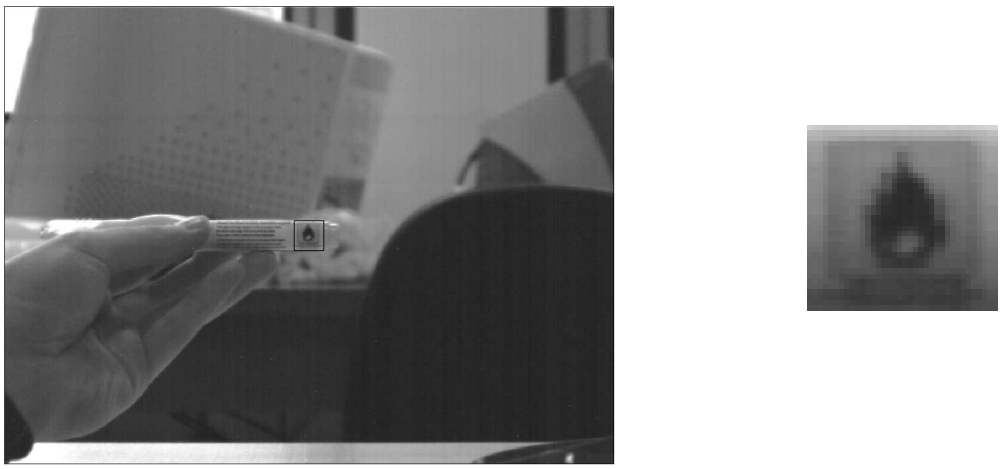


FIG. 7.8 – *À gauche : sélection du motif à suivre. À droite : motif sélectionné.*

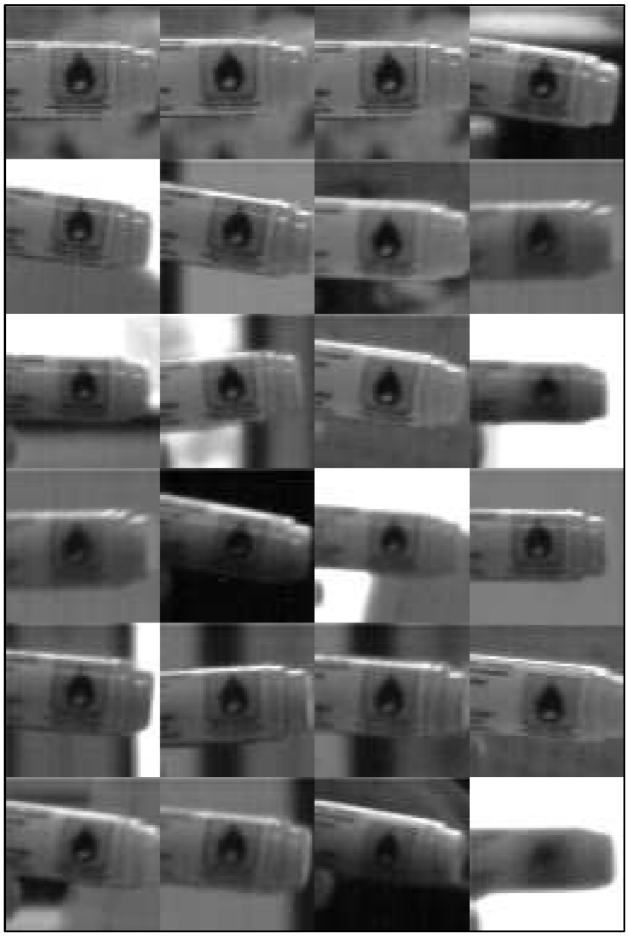


FIG. 7.9 – *Résultat du suivi. L'image reste centrée sur le motif, malgré les mouvements de celui-ci. Seule la zone d'intérêt autour du motif suivi est acquise.*

Les performances temporelles peuvent être améliorées jusqu'à environ 100 fps. Au delà, il serait nécessaire de diminuer le temps d'intégration des images, qui est de $10ms$ dans l'expérimentation réalisée. Ceci pourrait engendrer la nécessité de conditions d'éclairage contrôlées afin de conserver la bonne qualité des images, et du suivi par conséquent.

La méthode de suivi implémentée est un bon exemple d'application de vision active, et plus précisément de tâche de focalisation. En effet, les déplacements de la fenêtre de recherche dans le plan optique peuvent être comparés aux mouvements de la fovéa dans le mécanisme biologique de vision par saccades. Le résultat obtenu n'est pas une fenêtre statique où l'on viendra localiser l'objet suivi, mais plutôt une fenêtre mouvante constamment centrée sur celui-ci (comme l'on peut observer dans la figure 7.9). C'est l'équivalent d'un observateur qui suit un objet des yeux, en le gardant en permanence au centre de son champ visuel (fovéa).

Chapitre 8

Conclusion

"Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning."

"Ceci n'est pas la fin. Ce n'est même pas le commencement de la fin. Mais c'est, peut être, la fin du commencement."

Sir Winston Churchill (1874 - 1965), écrivain, historien et ancien premier ministre du Royaume-Uni. Discours prononcé en novembre 1942, après une victoire décisive des alliés contre l'Afrika Korps allemande en Égypte.

Les travaux rapportés dans ce manuscrit ont été réalisés au sein du groupe GRAVIR (Groupe d'Automatique : Vision et Robotique) au LASMEA (Laboratoire des Sciences et Matériaux pour l'Électronique et d'Automatique). Ils s'inscrivent dans la thématique "Systèmes de Perception", à cheval entre les axes de recherche "Capteurs Intelligents" et "Prototypage rapide d'applications".

La caméra intelligente SeeMOS a été utilisée comme support de développement et d'expérimentation. Cette plateforme hétérogène dédiée aux applications de vision active et précoce a été développée dans le cadre de la thèse de Pierre Chalimbaud [31]. Le travail ici présenté s'appuie en partie sur ses travaux.

L'objectif de cette thèse est d'apporter un certain nombre de solutions et contributions dans le domaine des caméras intelligentes (*smart cameras*), et plus particulièrement dans les techniques d'implémentation d'applications au sein de tels dispositifs.

Des contributions théoriques ont été apportées dans l'étude et la formalisation du concept de *smart camera*. Ces contributions se matérialisent notamment par la publication de deux chapitres dans l'ouvrage à distribution internationale *"Smart Cameras"* [48, 49, 51], édité par M. Ahmed N. Belbachir, et publié par les éditions Springer-Verlag (New York). Il s'agit du premier ouvrage de ce genre dédiée exclusivement aux caméras intelligentes.

Un certain nombre de contributions pratiques ont également été présentées, d'une part par le développement et l'implantation d'applications de vision sur la plateforme embarquée SeeMOS, et d'autre part par la publication de ces résultats dans plusieurs congrès internationaux [35, 110, 115, 36, 114]. Ces efforts d'implémentation ont abouti à la formalisation d'une méthodologie générale, dédiée aux plateformes du type caméra intelligente basées sur un composant reconfigurable FPGA. Les motivations, la formalisation, le développement et l'application de cette méthodologie générale sont les fils conducteurs du manuscrit ci-présent.

La première partie de ce manuscrit (chapitres 1 à 4) est consacrée à l'étude de la problématique et des motivations justifiant la réalisation de ce travail de recherche. En partant d'un bref rappel historique sur l'évolution de la vision artificielle, nous arrivons aux travaux ayant posé les bases du paradigme de la Vision Active. Par l'analyse des caractéristiques et des contraintes opérationnelles de ces tâches dites "actives", nous venons naturellement au concept de Caméra Intelligente, ou *Smart Camera* dans la terminologie anglaise. Nous nous intéressons ensuite aux différents aspects concernant ces dispositifs. Dans un premier temps les aspects matériels sont traités, par la description des technologies utilisées pour leur conception, et par la présentation de quelques exemples issus des milieux industriel et scientifique. En constatant la complexité et le haut niveau d'expertise requis par ces systèmes, nous nous concentrons enfin sur les différentes méthodes existantes permettant de faciliter le développement et l'implémentation d'applications au sein de ces plateformes.

La deuxième partie du manuscrit concerne plus directement les travaux réalisés au cours de cette thèse. Elle part du constat que, si des nombreux outils existent pour la description, la parallélisation et le partitionnement des applications de traitement du signal, peu de méthodes s'attaquent précisément aux aspects bas niveau relatifs au contrôle des composants électroniques mis en jeu. Or, dans le cas spécifique des plateformes embarquées basées sur FPGA, la gestion de ces éléments requiert un degré d'expertise important, et un effort de développement et de mise en oeuvre considérable. Il est également important de souligner que l'expertise en électronique numérique ne fait pas forcément partie de la palette de compétences de tout programmeur en vision artificielle.

La méthodologie développée dans ces travaux de recherche essaie de combler cette lacune : comment réaliser le pont entre d'une part l'algorithme et son architecture d'exécution, et d'autre part les composants hardware de la plateforme qui l'héberge. Dans un premier temps le concept méthodologique adopté est présenté (design à deux niveaux), suivi par la description des différents éléments et outils développés pour sa mise en oeuvre (langage de programmation, outil *assembleur*, modèles virtuel et soft-core). Nous démontrons ensuite comment la méthodologie a été appliquée à la plateforme SeeMOS, culminant dans des exemples expérimentaux dont le dernier est particulièrement représentatif des tâches de vision "actives" (stratégie d'acquisition *task-driven*, rétroaction au niveau du capteur et focalisation sur zone d'intérêt). Après tout, c'est bien par la vision active que tout avait commencé...

Un des points clefs de l'approche proposée est le découplage entre les aspects intrinsèquement liés à la plateforme matérielle, et ceux liés à l'application. Ceci se fait par l'introduction d'une couche intermédiaire, qui est matérialisée par un processeur soft-core simple de type RISC (CCU).

Du côté de la plateforme matérielle, un ensemble de pilotes dédiés est chargé de gérer les différents composants électroniques, en générant les signaux de synchronisation et contrôle nécessaires. Ces pilotes sont complètement indépendants de l'application.

Du côté de l'application, un ensemble de *Processing Elements* (PE's) est chargé d'implémenter les fonctions de traitement du signal constituant l'algorithme en question. Ces PE's sont indépendants de la plateforme matérielle cible.

Le pont entre les pilotes et les PE's ne se fait qu'au moment de l'exécution, en passant par la CCU qui organise le déroulement des opérations. Celle-ci peut être programmée au moyen d'un code assembleur simplifié, pouvant ne contenir que la description fonctionnelle de l'application (appels aux pilotes et PE's).

Ainsi, la méthodologie proposée dans cette thèse ne vise pas à remplacer ou concurrencer les nombreux outils de développement existants, mais plutôt à offrir une passerelle permettant d'appliquer leurs résultats à une plateforme reconfigurable *custom*, avec un effort de développement et de programmation moindre. Par

exemple, il est parfaitement envisageable d'utiliser en amont un outil de partitionnement et d'ordonnancement, afin d'optimiser l'implémentation de l'algorithme, et puis d'adapter ses résultats à une plateforme précise, en passant par l'approche méthodologique proposée ici. De la même façon, un traducteur *C-like to HDL* peut être utilisé afin de générer un PE câblé compatible avec le système.

Un autre point original exploré dans ces travaux est l'utilisation d'un modèle virtuel de la plateforme, créé en langage SLDL et permettant le prototypage rapide et le débogage *off-line*. Ceci apporte une très grande flexibilité au niveau du prototypage, permettant une variation progressive de la granularité de description. De plus, le langage utilisé (SpecC) est plus "convivial" que les HDL, et sa compilation plus rapide. Ainsi, le processus de développement et débogage gagne nettement en rapidité et fluidité, si comparé aux lentes et répétées synthèses VHDL nécessaires lors d'un développement direct "*on-board*".

Finalement, on peut ajouter que l'utilisation d'un modèle virtuel et du développement *off-line* renforce la notion de découplage entre aspects matériels et applicatifs. L'application peut donc être entièrement développée sur la plateforme virtuelle, sans être confrontée aux particularités d'une quelconque plateforme cible "réelle" (sauf, bien évidemment, au moment de son implémentation finale). Ceci permet d'éviter les incertitudes liées au bon fonctionnement de la plateforme matérielle au moment des tests.

Bien évidemment, les travaux présentés dans ce manuscrit laissent des nombreuses voies ouvertes pour des améliorations et développements futurs. Une de ces voies est le développement d'un compilateur capable de générer le code assembleur pour la CCU, à partir d'une description de l'application dans un langage haut-niveau à définir (textuel ou graphique). Cette description pourrait contenir des directives explicites de parallélisation, telles que les langages SLDL, ou utiliser un mécanisme d'inférence automatique des parallélismes, par la création d'un graphe des dépendances de données (comme par exemple la méthodologie SynDEx).

D'autres aspects relatifs à l'Adéquation Algorithme-Architecture peuvent être également explorés, et certains travaux sont déjà en cours au sein du laboratoire, comme par exemple des recherches concernant le partitionnement DSP/FPGA, et le "mapping" des fonctions au sein de ces dispositifs.

Une autre voie laissée ouverte est le paramétrage automatique de l'architecture de la CCU, par le profilage du code assembleur à exécuter (définition du nombre de registres, profondeur de la pile, taille de la mémoire programme, largeur des bus, etc...). L'ensemble de ces travaux pourrait converger à terme vers un environnement de développement unique, englobant la totalité du processus de design. Cet environnement devra être capable de générer, à partir d'une description haut-niveau et d'un ensemble de librairies, tous les codes source nécessaires pour l'implémentation de l'application au sein d'une plateforme donnée (code HDL pour l'instantiation dans le FPGA, code C pour le DSP, et code assembleur pour la CCU).

Bibliographie

[1] Guillaume Libri : *Histoire des Sciences Mathématiques en Italie, depuis la renaissance des lettres jusqu'a la fin du dix-septieme siècle*, volume 4. Jules Renouard et Cie, 1841.

[2] Giovanni Battista Venturi : *Essai sur les ouvrages physico-mathématiques de Léonard de Vinci*. Duprat, 1797.

[3] Donald A. Glaser : Some Effects of Ionizing Radiation on the Formation of Bubbles in Liquids. *Physical Review*, 87(4):665, 1952.

[4] A. M. Turing : Computing Machinery and Intelligence. *Mind*, 59:433–460, 1950.

[5] David Marr : *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. W. H. Freeman, 1982.

[6] Cornelia Fermuller et Yiannis Aloimonos : What is computed by structure from motion algorithms? *In Proc. European Conference on Computer Vision (ECCV)*, pages 359–375, 1998.

[7] Richard Hartley et Andrew Zisserman : *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2003.

[8] B. K. P. Horn : Obtaining Shape from Shading Information. *In The Psychology of Computer Vision*, pages 115–155, 1975.

[9] E. Prados et O. D. Faugeras : Shape from Shading: A Well-Posed Problem? *In International Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages 870–877, 2005.

[10] Olivier Baujard et Catherine Garbay : KISS : Un système de vision multi-agents. *In 7éme Congrès Reconnaissance des Formes et Intelligence Artificielle (RFIA)*, pages 89–98, 1989.

[11] Valéry Lefèvre, Yann Pollet, Sylvie Philipp et Sylvie Brunessaux : Un système multi-agents pour la fusion de données en analyse d'images. *Traitement du Signal*, 13(1):99–111, 1996.

[12] Yiannis Aloimonos, Isaac Weiss et Amit Bandyopadhyay : Active Vision. *In 1st International Conference on Computer Vision (ICCV)*, pages 35–54, 1987.

[13] Ruzena Bajcsy : Active Perception. *Proceedings of the IEEE, Special issue on Computer Vision*, 76(8):996–1005, 1988.

[14] Ruzena Bajcsy et Mario Campos : Active and exploratory perception. *CVGIP : Image Understanding (Computer Vision Graphics and Image Processing)*, 56(1):31–40, 1992.

[15] Dana H. Ballard : Reference Frames for Animate Vision. *In International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1635–1641, 1989.

[16] Dana H. Ballard : Animate vision. *Artificial Intelligence Journal*, 48:57–86, 1991.

[17] Yiannis Aloimonos : Purposive and Qualitative active vision. *In Proc. 10th International Conference on Pattern Recognition (ICPR)*, volume 1, pages 346–360, 1990.

[18] Ruzena Bajcsy : Active Perception vs. Passive Perception. *In Proceedings of the 3rd IEEE Workshop on Computer Vision: Representation and Control*, pages 55–62, 1985.

[19] J. Santos-Victor, F. Trigt et J. Sentieiro : MEDUSA - A Stereo Head for Active Vision. *In Proc. International Symposium on Inteligent Robotic Systems*, 1994.

[20] R. James Firby, Roger E. Kahn, Peter N. Prokopowicz et Michael J. Swain : An architecture for vision and action. *In 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 72–79, 1995.

[21] J. Findlay et I. Gilchrist : *Visual attention: the active vision perspective*, chapitre 5 de l'ouvrage "Vision and attention", pages 85–106. Springer-Verlag, 2001.

[22] F. Wörgötter, N. Krüger, N. Pugeault, D. Calow, M. Lappe, K. Pauwels, M. Van Hulle, S. Tan et A. Johnston : Early Cognitive Vision: Using Gestalt-Laws for Task-Dependent, Active Image-Processing. *Natural Computing*, 3(3):293–321, 2004.

[23] B. Marsh, C. Brown, T. LeBlanc, M. Scott, T. Becker, P. Das, J. Karlsson et C. Quiroz : Operating system support for animate vision. *Journal of Parallel and Distributed Computing*, 15(2):103–117, 1992.

[24] Jeffrey A. Fayman, Ehud Rivlin et Henrik I. Christensen : The Active Vision Shell. Rapport technique, Israel Institute of Technology and Aalborg University, Danemark, 1995.

[25] G. Granlund : Does vision inevitably have to be active. *In Scandinavian Conference on Image Analysis (SCIA)*, 1999.

[26] J.K. Tsotsos : On the Relative Complexity of Active vs. Passive Visual Search. *Internatinal Journal of Computer Vision (IJCV)*, 7(2):127–141, 1992.

[27] A. Andreopoulos et J. K. Tsotsos : Active Vision for Door Localization and Door Opening using Playbot: A Computer Controlled Wheelchair for People with Mobility Impairments. *In Canadian Conference on Computer and Robot Vision (CRV)*, pages 28–30, 2008.

[28] Eric Marchand : *Stratégies de perception par vision active pour la recons-*

truction et l'exploration de scènes statiques. Thèse de doctorat, Université de Rennes, France, 1996.

[29] T. Viéville : *A few steps towards 3D Active Vision*, volume 33. Springer Series in Information Sciences, 1997.

[30] A. Blake et A. Yuille : *Active Vision.* MIT Press, 1993.

[31] Pierre Chalimbaud : *Conception d'une plateforme d'implémentation matérielle dédiée aux systèmes de vision active basés sur un imageur CMOS.* Thèse de doctorat, Université Blaise-Pascal, Clermont-Ferrand, France, 2004.

[32] Le cerveau à tous les niveaux. Site web, Instituts de Recherche en Santé du Canada (IRSC) : Instituts des neurosciences, de la santé mentale et des toxicomanies.
http://lecerveau.mcgill.ca/.

[33] Hans-Werner Hunziker : *Im Auge des Lesers: foveale und periphere Wahrnehmung - vom Buchstabieren zur Lesefreude (L'oeil du lecteur: les perceptions périphériques et fovéales - comment arriver à la joie de lire).* Transmedia Stäubli Verlag, Zürich, 2006.

[34] P. Chalimbaud et F. Berry : Contrast Optimization in a Multi-Windowing Image Processing Architecture. *In IAPR Conference on Machine Vision Applications (MVA)*, 2005.

[35] F. Dias Real, P. Chalimbaud, F. Berry, J. Sérot et F. Marmoiton : Embedded Early Vision systems: implementation proposal and Hardware Architecture. *In Cognitive Systems with Interactive Sensors (COGIS)*, 2006.

[36] F. Dias Real, F. Berry, J. Sérot et F. Marmoiton : Hardware, Design and Implementation Issues on a FPGA-based Smart Camera. *In International Conference on Distributed Smart Cameras (ICDSC)*, 2007.

[37] Stefan Treue : Visual attention: the where, what, how and why of saliency. *Current opinion in neurobiology*, 13(4):428–432, 2003.

[38] C. Koch et S. Ullman : Shifts in selective visual attention: towards the underlying neural circuitry. *Human Neurobiology*, 4:219–227, 1985.

[39] L. Itti, C. Koch et E. Niebur : A Model of Saliency-Based Visual Attention for Rapid Scene Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(11):1254–1259, 1998.

[40] K. Rapantzikos et N. Tsapatsoulis : On the implementation of visual attention architectures. *In Proceedings of the workshop on Attention Control Architectures*, pages 245–250, 2003.

[41] Min Chul Park, Kyung Joo Cheoi et Takayuki Hamamoto : A Smart Image Sensor with Attention Modules. *In International Workshop on Computer Architecture for Machine Perception (CAMP)*, pages 46–51, 2005.

[42] A. L. Yarbus : *Eye Movements and Vision.* New York: Plenum Press, 1967.

[43] Jonathan D. Nelson, Garrison W. Cottrell, Javier R. Movellan et Martin I. Sereno : Yarbus lives: a foveated exploration of saccadic eye movement

(abstract). *Journal of Vision*, 4(8):741, 2004.
http://mplab.ucsd.edu/∼jnelson/foveation.html.

[44] W. Wong et R. I. Hornsey : Design of an Electronic Saccadic Imaging System. *In Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 2227–2230, 2004.

[45] Pelegrin Camacho, Fabian Arrebola et Franscisco Sandoval : Multiresolution Sensors with Adaptive Structure. *In Conference of the IEEE Industrial Electronics Society (IECON)*, volume 2, pages 1230–1235, 1998.

[46] Shimon Ullman : Visual routines. *Cognition*, 18:97–159, 1984.

[47] W. Wolf, B. Ozer et T. Lu : Smart Cameras as Embedded Systems. *IEEE Computer*, 35(9):48–53, Sep 2002.

[48] A. N. Belbachir, éditeur. *Smart Cameras*. Springer-Verlag New York, 2010.

[49] Yu Shi et Fabio Dias Real : *Smart Cameras*, chapitre 2 - Smart Cameras: Fundamentals and Classification, pages 19–34. Springer-Verlag New York, 2010.

[50] M. Bramberger, A. Doblander, A. Maier, B. Rinner et H. Schwabach : Distributed Embedded Smart Cameras for Surveillance Applications. *IEEE Computer*, 39(2):68–75, Feb 2006.

[51] Fabio Dias Real et François Berry : *Smart Cameras*, chapitre 3 - Smart Cameras: Technologies and Applications, pages 35–50. Springer-Verlag New York, 2010.

[52] P. Chalimbaud, F. Marmoiton et F. Berry : Towards an Embedded Visuo-Inertial Smart Sensor. *International Journal of Robotics Research (IJRR)*, 26(6):537–546, 2007.

[53] A. Wilson : Smart camera monitors traffic. *Vision Systems Design*, 13(7), Jul 2008.

[54] H. Zhou, M. Taj et A. Cavallaro : Audiovisual Tracking using STAC sensors. *In International Conference on Distributed Smart Cameras (ICDSC)*, 2007.

[55] W. Wolf et P. Cook : Smart Cameras and Microphones. *In NSF Workshop on Cyber-Physical Systems*, 2006.

[56] Imène Berkhermi, Amine Benkhelifa, Daniel Chillet, Sébastien Pillement, Jean-Christophe Prevotet et François Verdier : Modélisation niveau système de SoC reconfigurable. *In SympAAA*, 2005.

[57] A. Rowe, A. Goode, D. Goel et I. Nourbakhsh : CMUcam3: An Open Programmable Embedded Vision Sensor. Technical Report RI-TR-07-13, Carnegie Mellon Robotics Institute, 2007.

[58] S. Hengstler, D. Prashanth, S. Fong et H. Aghajan : MeshEye: A Hybrid-Resolution Smart Camera Mote for Applications in Distributed Intelligent Surveillance. *In International Symposium on Information Processing in Sensor Networks (IPSN)*, pages 360–369, 2007.

[59] M. Bramberger, J. Brunner, B. Rinner et H. Schwabach : Real-Time Video Analysis on an Embedded Smart Camera for Traffic Surveillance. *In Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 174–181, 2004.

[60] R. Mosqueron, J. Dubois et M. Paindavoine : High-Speed Smart Camera with High Resolution. *EURASIP Journal on Embedded Systems*, page 16 pages, 2007. doi:10.1155/2007/24163.

[61] R. Kleihorst, B. Schueler, A. Danilin et M. Heijligers : Smart Camera Mote with High Performance Vision System. *In Workshop on Distributed Smart Cameras (DSC)*, 2006.

[62] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir et T. Arslan : The Reconfigurable Instruction Cell Array. *In IEEE Transactions on very large scale integration (VLSI) systems*, volume 16, pages 75–85, 2008.

[63] Branislav Kisacanin : Examples of Low-Level Computer Vision on Media Processors. *In IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005.

[64] R. Huizen : FPGA/DSP hybrid architectures: Satisfying the reconfigurability requirements of the military. *Military Embedded Systems (MES)*, Sep, 2007.

[65] FPGA, CPLD and ASIC from Altera. Site web. *http://www.altera.com.*

[66] FPGA and CPLD Solutions from Xilinx, Inc. Site web. *http://www.xilinx.com.*

[67] Jan-Willem van de Waerdt : *The TM3270 Media-processor.* Thèse de doctorat, Delft University of Technology, Pays-Bas, 2006.

[68] Harald Vranken : Debug Facilities in the TriMedia CPU64 Architecture. *Journal of Electronic Testing : Theory and Applications*, 16:301–308, 2000.

[69] The CMUcam Vision Sensors. The Robotics Institute - Carnegie Mellon University. Site web. *http://www.cs.cmu.edu/∼cmucam/.*

[70] SmartCam Project. Graz University of Technology - Institute for Technical Informatics. Site web. *http://www.iti.tu-graz.ac.at/en/research/smartcam/site/.*

[71] B. Rinner, M. Jovanovic et M. Quaritsch : Embedded Middleware on Distributed Smart Cameras. *In Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 1381–1384, 2007.

[72] Markus Quaritsch, Markus Kreuzthaler, Bernhard Rinner, Horst Bischof et Bernhard Strobl : Autonomous Multicamera Tracking on Embedded Smart Cameras. *EURASIP Journal on Embedded Systems*, page 10 pages, 2007. doi:10.1155/2007/92827.

[73] Jan van der Horst : Development and implementation of a real-time stereo camera. Mémoire de Master, Delft University of Technology, Pays-Bas, 2006.

[74] D. Bauer, A. Belbachir, N. Donath, G. Gritsch, M. Litzenberger B. Kohn, C. Posch, P. Schön et S. Schraml : Embedded Vehicle Speed Estimation System Using an Asynchronous Temporal Contrast Vision Sensor. *EURASIP Journal on Embedded Systems*, page 12 pages, 2007. doi:10.1155/2007/82174.

[75] Smart Optical Sensors, Smart Systems, Austrian Research Centers GmbH. Site web. *http://www.smart-systems.at/products/products_smart_optical_sensors_en.html*.

[76] L. Albani, P. Chiesa, D. Covi, G. Pedegani, A. Sartori et M. Vatteroni : VISoc : A Smart Camera SoC. *In Proceedings of the 28th European Solid-State Circuits Conference*, pages 367–370, Sep 2002.

[77] B. Flinchbaugh : *Real-Time Vision for Human-Computer Interaction*, chapitre Smart Cameras Systems Technology Roadmap, pages 285–297. Springer US, 2005.

[78] SICK-IVP Smart Cameras. Site web. *http://www.sickivp.se/sickivp/products/smart_cameras/en.html*.

[79] Anthony Rowe : *CMUcam2GUI Overview Manual*. Carnegie Mellon University, 2003. *http://www.cs.cmu.edu/∼cmucam2/CMUcam2GUI_overview.pdf*.

[80] Thierry Grandpierre et Yves Sorel : From Algorithm and Architecture Specification to Automatic Generation of Distributed Real-Time Executives: a Seamless Flow of Graphs Transformations. *In Proceedings of First ACM and IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 123–132, 2003.

[81] SynDEx: System-Level CAD Software for Distributed Real-Time Embedded Systems. Site web. *http://www-roc.inria.fr/syndex/*.

[82] Florent Berthelot, Fabienne Nouvel et Dominique Houzet : Design methodology for dynamically reconfigurable systems. *In Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA)*, 2005.

[83] C. Hylands, E. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, Y. Zhao et H. Zheng : Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M03/25, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 2003.

[84] Lars Wernli : Design and Implementation of a Code Generator for the Cal Actor Language. Rapport technique UCB/ERL M02/5, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 2002.

[85] R. Thavot, R. Mosqueron, M. Alisafaee, C. Lucarz, M. Mattavelli, J. Dubois et V. Noel : Dataflow design of a co-processor architecture for image processing. *In Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2008.

[86] Jocelyn Sérot et Dominique Ginhac : Skeletons for parallel image processing: an overview of the SKIPPER project. *Parallel Computing*, 28(12):1685–1708, 2002.

[87] K. Benkrid, D. Crookes, J. Smith et A. Benkrid : High Level Programming for FPGA Based Image and Video Processing Using Hardware Skeletons. *In Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 219–226, 2001.

[88] Gary Spivey, Shuvra S. Bhattacharyya et Kazuo Nakajima : Logic Foundry: A Rapid Prototyping Tool for FPGA-based DSP Systems. Technical Report UMIACS-TR-2002-29, Institute for Advanced Computer Studies, University of Maryland, 2002.

[89] L. Kaouane, M. Akil, T. Grandpierre et Y. Sorel : A methodology to implement real-time applications on reconfigurable circuits. *In Proceedings of International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2003.

[90] P. Niang, T. Grandpierre, M. Akil et Y. Sorel : SynDEx-IC : un Environnement Logiciel pour l'Implantation Optimisée d'Applications temps réel sur Circuits Reconfigurables. *In Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA)*, 2005.

[91] C-to-FPGA Tools from Impulse Accelerated Technologies. Site web. *http://www.impulseaccelerated.com/products_universal.htm*.

[92] Ian Page : Hardware-software Co-synthesis Research at Oxford, 1996.

[93] DK Design Suite - Rapid path from C to FPGA :: Handel-C :: FPGA. Site web. *www.agilityds.com/products/c_based_products/dk_design_suite/default.aspx*.

[94] FPGA C Compiler. Site web. *http://fpgac.sourceforge.net/*.

[95] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden et D. Zaretsky : A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems. *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 39–48, 2000. doi:10.1109/FPGA.2000.903391.

[96] C to Verilog — Circuit design automation. Site web. *http://www.c-to-verilog.com/index.html*.

[97] JHDL FPGA CAD TOOLS – Brigham Young University. Site web. *http://www.jhdl.org/*.

[98] Lionel Lelong, Guy Motyl, Nathalie Bochard et Gérard Jacquet : Architecture SoC-FPGA pour la mesure temps réel par traitement d'images. *In Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA)*, 2005.

[99] Jean Pierre DÉRUTIN, Lionel DAMEZ, Adrien DESPORTES et Jose Luis Lazaro GALILEA : Design of a Scalable Network of Communicating Soft Processors on FPGA. *In International Workshop on Computer Architecture for Machine Perception and Sensing (CAMP)*, pages 184–189, 2006. doi:10.1109/CAMP.2007.4350378.

[100] Anders DELLSON : Programming FPGAs for High-Performance Computing Acceleration. *Xcell Journal*, Issue 55, 2005. *http://www.xilinx.com/publications/xcellonline/xcell_55/xc_mitrion55.htm*.

[101] V. BROST, F. YANG et M. PAINDAVOINE : Un modèle VHDL du DSP C6201 avec un jeu d'instructions variable. *In Journées Francophones sur l'Adéquation Algorithme Architecture (JFAAA)*, 2005.

[102] Domingo Torres LUCIO : *Elaboration et Validation de LAPMAM : processeur parallèle SIMD/MIMD dédié au traitement bas et moyen niveau d'images*. Thèse de doctorat, Université Henri Poincaré, Nancy, France, 1999.

[103] A. GUPTA et R. DÖMER : System Design of Digital Camera Using SpecC. Technical Report CECS-04-32, Center for Embedded Computer Systems, University of California, Irvine, 2004.

[104] Open SystemC Initiative (OSCI). Site web. *http://www.systemc.org/home/*.

[105] The SpecC System. Site web. *http://www.cecs.uci.edu/∼specc/*.

[106] Rainer DÖMER : *The SpecC Language (Tutorial)*. Center for Embedded Computer Systems, University of California, Irvine, 2001. *http://www.cecs.uci.edu/∼specc/language.pdf*.

[107] Rainer DÖMER, Andreas GERSTLAUER et Daniel GAJSKI : *SpecC Language Reference Manual - Version 2.0*. SpecC Technology Open Consortium, 2002. *http://www.cecs.uci.edu/∼specc/SpecC_LRM_20.pdf*.

[108] Andreas GERSTLAUER : *The SpecC Methodology (Tutorial)*. Center for Embedded Computer Systems, University of California, Irvine, 2001. *http://www.cecs.uci.edu/∼specc/methodology.pdf*.

[109] Jean-Claude HEUDIN et Christian PANETTO : *Les Architectures RISC - Théorie et pratique des ordinateurs à jeu d'instructions réduits*. Editions Dunod, 1990.

[110] F. DIAS REAL, F. BERRY, J. SÉROT et F. MARMOITON : Configurable Window-Based Processing Element for Image Processing on Smart Cameras. *In IAPR Conference on Machine Vision Applications (MVA)*, 2007.

[111] C. T. HUITZIL et M. A. ESTRADA : FPGA-Based Configurable Systolic Architecture for Window-Based Image Processing. *EURASIP Journal on Applied Signal Processing*, 7:1024–1034, 2005.

[112] H. YU et M. LEESER : Optimizing data intensive window-based image processing on reconfigurable hardware boards. *In IEEE Workshop on Signal Processing Systems Design and Implementation*, pages 491–496, 2005.

[113] H. Pourreza, M. Rahmati et F. Behazin : Weighted multiple bit-plane matching, a simple and efficient matching criterion for electronic digital image stabilizer application. *In 6th International Conference on Signal Processing*, volume 2, pages 957–960, 2002.

[114] M. Abid, F. Dias Real, F. Berry et J. Sérot : Harnessing a multi-sensor FPGA-based Smart Camera: a virtual processor-based approach. *In Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2009.

[115] F. Dias Real, F. Berry, J. Sérot et F. Marmoiton : Smart Camera with Embedded Tracking Algorithm. *In 6th IFAC Symposium on Intelligent Autonomous Vehicles (IAV)*, 2007.

Annexes

Annexe A - Code HDL généré par un traducteur “C to Verilog”

Code d’application en langage C :

```
//returns the mean between two values
static inline unsigned int mean(unsigned int input1, unsigned int
input2) {

  unsigned int mean_v = 0;
  mean_v = input1 + input2;
  mean_v = mean_v/2;

  return mean_v;
}

void my_main(unsigned int *A, unsigned int *B, unsigned int *C) {
    unsigned char i;
    for (i=0; i<100; i++){
       C[i] = mean(A[i], B[i]);
    }
}
```

Code généré en langage Verilog :

```
module _Z7my_mainPjS_S_  (clk, reset, rdy,// control
    mem_A_out0, mem_A_in0, mem_A_addr0, mem_A_mode0, // memport for: A
    mem_B_out0, mem_B_in0, mem_B_addr0, mem_B_mode0, // memport for: B
    mem_C_out0, mem_C_in0, mem_C_addr0, mem_C_mode0, // memport for: C
    p_A, p_B, p_C, return_value); // params
 input wire clk;
 input wire reset;
 output rdy;
 reg rdy;
 output return_value;
 reg return_value;
 input [15:0] p_A;
 input [15:0] p_B;
 input [15:0] p_C;
input wire [31:0] mem_A_out0; output reg [31:0] mem_A_in0; output
reg [15:0] mem_A_addr0; output reg mem_A_mode0; input wire [31:0]
mem_B_out0; output reg [31:0] mem_B_in0; output reg [15:0]
mem_B_addr0; output reg mem_B_mode0; input wire [31:0] mem_C_out0;
output reg [31:0] mem_C_in0; output reg [15:0] mem_C_addr0; output
reg mem_C_mode0;


 reg [7:0] i_incrementVal71;   /*local var*/
 reg [7:0] i_incrementVal26;   /*local var*/
 reg [7:0] i_incrementVal28;   /*local var*/
 reg [7:0] i_incrementVal27;   /*local var*/
 reg [7:0] i_incrementVal;   /*local var*/
 reg [7:0] i_cloned10___0___;   /*local var*/
 reg [7:0] i_cloned10___1___;   /*local var*/
 reg [31:0] i_tmp11___0___;   /*local var*/
 reg [31:0] i_tmp6___0___;   /*local var*/
 reg i_exitcond9___0___;   /*local var*/
 reg [31:0] i_tmp11___1___;   /*local var*/
 reg [31:0] i_tmp6___1___;   /*local var*/
 reg [31:0] i_tmp3_i___0___;   /*local var*/
 reg [31:0] i_tmp11___2___;   /*local var*/
 reg [31:0] i_tmp6___2___;   /*local var*/
 reg [31:0] i_tmp3_i___1___;   /*local var*/
 reg [7:0] i_incrementVal143;   /*local var*/
 reg [7:0] i_incrementVal146;   /*local var*/
 reg [31:0] i_tmp3_i;   /*local var*/
 reg [7:0] i_indvar_next8;   /*local var*/
 reg i_exitcond9;   /*local var*/
 reg [31:0] i_tmp11;   /*local var*/
 reg [31:0] i_tmp6;   /*local var*/
 reg [7:0] i_cloned10;   /*local var*/
 reg i_gluePipelinedLoop157;   /*phi var*/
 reg [7:0] i_gluePipelinedLoop156;   /*phi var*/
 reg [15:0] p_gluePipelinedLoop154;   /*phi var*/
 reg [31:0] i_gluePipelinedLoop153;   /*phi var*/
 reg [31:0] i_gluePipelinedLoop152;   /*phi var*/
 reg [31:0] i_gluePipelinedLoop151;   /*phi var*/
 reg [15:0] p_gluePipelinedLoop150;   /*phi var*/
 reg [31:0] i_gluePipelinedLoop149;   /*phi var*/
 reg [15:0] p_gluePipelinedLoop148;   /*phi var*/
 reg [31:0] i_gluePipelinedLoop147;   /*phi var*/
 reg [31:0] i_gluePipelinedLoop145;   /*phi var*/
 reg [7:0] i_i_01_0;   /*phi var*/
```

```
 // Number of states:10
 reg [3:0] eip;
 parameter entry0 = 4'd0;
 parameter entry1 = 4'd1;
 parameter entry2 = 4'd2;
 parameter entry3 = 4'd3;
 parameter entry4 = 4'd4;
 parameter entry5 = 4'd5;
 parameter PipelinedLoop0 = 4'd6;
 parameter PipelinedLoop1 = 4'd7;
 parameter PipelinedLoop2 = 4'd8;
 parameter return0 = 4'd9;

// Assign part (0)


always @(posedge clk)
 begin
  if (reset)
   begin
    $display("@hard reset");
    eip<=0;
    rdy<=0;
   end


// Datapath
i_incrementVal71 <= (0)+(0); i_incrementVal26 <= (1)+(0);
i_incrementVal28 <= (2)+(0); i_incrementVal27 <= (3)+(0);
i_incrementVal <= (4)+(0); i_cloned10___0___ <=
i_incrementVal71+(1); i_cloned10___1___ <= i_incrementVal26+(1);
i_exitcond9___0___ <= (i_cloned10___0___ == (100)); i_tmp3_i___0___
<= i_tmp11___0___+i_tmp6___0___; i_tmp3_i___1___ <=
i_tmp11___1___+i_tmp6___1___; i_incrementVal143 <= (5)+i_i_01_0;
i_incrementVal146 <= (2)+i_i_01_0; i_tmp3_i <=
i_gluePipelinedLoop151+i_gluePipelinedLoop149; i_indvar_next8 <=
i_i_01_0+(1); i_exitcond9 <= (i_gluePipelinedLoop156 == (100));
i_cloned10 <= i_incrementVal146+(1);


// Control
case (eip) entry0: begin
    eip <= entry1;
end entry1: begin
    mem_A_mode0 <= 0;
    mem_A_addr0 <= (p_A + (i_incrementVal71));
    mem_B_mode0 <= 0;
    mem_B_addr0 <= (p_B + (i_incrementVal71));
    eip <= entry2;
end entry2: begin
    i_tmp11___0___ <= mem_A_out0;
    mem_A_mode0 <= 0;
    mem_A_addr0 <= (p_A + (i_incrementVal26));
    i_tmp6___0___ <= mem_B_out0;
    mem_B_mode0 <= 0;
    mem_B_addr0 <= (p_B + (i_incrementVal26));
    eip <= entry3;
end entry3: begin
    i_tmp11___1___ <= mem_A_out0;
    mem_A_mode0 <= 0;
    mem_A_addr0 <= (p_A + (i_incrementVal28));
    i_tmp6___1___ <= mem_B_out0;
    mem_B_mode0 <= 0;
    mem_B_addr0 <= (p_B + (i_incrementVal28));
    eip <= entry4;
```

```
end entry4: begin
    i_tmp11___2___ <= mem_A_out0;
    i_tmp6___2___ <= mem_B_out0;
    eip <= entry5;
end entry5: begin
            i_gluePipelinedLoop157 <= i_exitcond9___0___;
        i_gluePipelinedLoop156 <= i_cloned10___1___;
        p_gluePipelinedLoop154 <= (p_C + (i_incrementVal71));
        i_gluePipelinedLoop153 <= (((i_tmp3_i___0___) >> ((1))));
        i_gluePipelinedLoop152 <= i_tmp3_i___1___;
        i_gluePipelinedLoop151 <= i_tmp11___2___;
        p_gluePipelinedLoop150 <= (p_A + (i_incrementVal27));
        i_gluePipelinedLoop149 <= i_tmp6___2___;
        p_gluePipelinedLoop148 <= (p_B + (i_incrementVal27));
        i_gluePipelinedLoop147 <= (i_incrementVal26);
        i_gluePipelinedLoop145 <= (i_incrementVal);
        i_i_01_0 <= (0);
        eip <= PipelinedLoop0;
end PipelinedLoop0: begin
    mem_A_mode0 <= 0;
    mem_A_addr0 <= p_gluePipelinedLoop150;
    mem_B_mode0 <= 0;
    mem_B_addr0 <= p_gluePipelinedLoop148;
    mem_C_in0 <= i_gluePipelinedLoop153;
    mem_C_mode0 <= 1;
    mem_C_addr0 <= p_gluePipelinedLoop154;
    eip <= PipelinedLoop1;
end PipelinedLoop1: begin
    i_tmp11 <= mem_A_out0;
    i_tmp6 <= mem_B_out0;
    mem_C_mode0 <= 0;
    eip <= PipelinedLoop2;
end PipelinedLoop2: begin
    if (i_gluePipelinedLoop157) begin
        eip <= return0;
    end else begin
        i_gluePipelinedLoop157 <= i_exitcond9;
        i_gluePipelinedLoop156 <= i_cloned10;
        p_gluePipelinedLoop154 <= (p_C + i_gluePipelinedLoop147);
        i_gluePipelinedLoop153 <= (((i_gluePipelinedLoop152) >> ((1))));
        i_gluePipelinedLoop152 <= i_tmp3_i;
        i_gluePipelinedLoop151 <= i_tmp11;
        p_gluePipelinedLoop150 <= (p_A + i_gluePipelinedLoop145);
        i_gluePipelinedLoop149 <= i_tmp6;
        p_gluePipelinedLoop148 <= (p_B + i_gluePipelinedLoop145);
        i_gluePipelinedLoop147 <= (i_incrementVal146);
        i_gluePipelinedLoop145 <= (i_incrementVal143);
        i_i_01_0 <= i_indvar_next8;
        eip <= PipelinedLoop0;
    end
end return0: begin
     rdy <= 1;
 return_value <= 0;
 $finish();
end
 endcase //eip
end //always @(..)

endmodule
```

Annexe B - Programme assembleur d'acquisition en mode caméra rapide

```
            // début du programme

            //première intégration
000         LOAD INT_TIME #20000        //temps d'intégration de 1ms
001         SET START_INTEG
002         WAIT INTEGRATING 1
003         RESET START_INTEG

            //chargement des registres d'acquisition
004         LOAD ADDRESS_X #1094        //1024 + size_x/2, pour centrer la fenêtre
005         LOAD ADDRESS_Y #1094        //1024 + size_y/2,      dans l'image
006         LOAD SIZE_X #140            //images de taille 140 x 140
007         LOAD SIZE_Y #140
008         LOAD AD_BASE_REC #0

            //chargement des registres pour l'envoi des données
009         LOAD AD_BASE_BURST #0
010         LOAD NUMBER_BURST #19600

            //début de la boucle
011         WAIT INTEGRATING 0      //attente de la fin de l'intégration 1

            //début de l'acquisition 1
012         GIVE IMG_SENSOR MEM_1
013         SET IMG_REC
014         SET START_ACQ
015         WAIT ACQUIRING 1
016         RESET START_ACQ

            //début de l'intégration 2
017         SET START_INTEG
018         WAIT INTEGRATING 1
019         RESET START_INTEG

            //attente de la fin de l'acquisition 1
020         WAIT ACQUIRING 0
021         RESET IMG_REC
022         GET MEM_1

            //attente de la fin de l'envoi de données 2 vers le host
023         WAIT BURSTING 0
024         GET MEM_2

            //envoi de données 1 vers le host
025         GIVE BURST MEM_1
026         SET START_BURST
027         WAIT BURSTING 1
028         RESET START_BURST

            //attente de la fin de l'intégration 2
029         WAIT INTEGRATING 0

            //début de l'acquisition 2
030         GIVE IMG_SENSOR MEM_2
031         SET IMG_REC
032         SET START_ACQ
033         WAIT ACQUIRING 1
```

```
034          RESET START_ACQ

             //début de l'intégration 1
035          SET START_INTEG
036          WAIT INTEGRATING 1
037          RESET START_INTEG

             //attente de la fin de l'acquisition 2
038          WAIT ACQUIRING 0
039          RESET IMG_REC
040          GET MEM_2

             //attente de la fin de l'envoi de données 1 vers le host
041          WAIT BURSTING 0
042          GET MEM_1

             //envoi de données 2 vers le host
043          GIVE BURST MEM_2
044          SET START_BURST
045          WAIT BURSTING 1
046          RESET START_BURST

047          JUMP $11    //retour au début de la boucle
```

Annexe C - Programme assembleur d'acquisition synchronisée d'images et mesures inertielles

```
        // début du programme

        // intégration
000     LOAD INT_TIME #200000       //temps d'intégration de 10ms
001     SET START_INTEG
002     WAIT INTEGRATING 1
003     RESET START_INTEG

        // chargement des registres de la centrale inertielle
004     LOAD AD_BASE_INERT #0

        // chargement des registres d'acquisition
005     LOAD ADDRESS_X #2024        //1024 + size_x/2
006     LOAD ADDRESS_Y #1274        //1024 + size_y/2
007     LOAD SIZE_X #2000
008     LOAD SIZE_Y #500
009     LOAD AD_BASE_REC #0

        // attente de la fin de l'intégration
010     WAIT INTEGRATING 0

        // début de l'acquisition inertielle
011     GIVE INERT MEM_3
012     SET INERT_REC

        // début de l'acquisition
013     GIVE REC MEM_1
014     SET IMG_REC
015     SET START_ACQ
016     WAIT ACQUIRING 1
017     RESET START_ACQ

        // attente de la fin de l'acquisition
018     WAIT ACQUIRING 0
019     RESET IMG_REC
020     GET MEM_1

        // début de la boucle

        // début de la deuxième intégration
021     LOAD INT_TIME #200000        //temps d'intégration de 10ms
022     SET START_INTEG
023     WAIT INTEGRATING 1
024     RESET START_INTEG

        // chargement des registres et envoi des données image 1 vers le host
025     GIVE BURST MEM_1
026     LOAD AD_BASE_BURST #0
027     LOAD NUMBER_BURST #1000000
028     SET START_BURST
029     WAIT BURSTING 1
030     RESET START_BURST

        // attente de la fin de la deuxième intégration
031     WAIT INTEGRATING 0

        // arrête l'acquisition inertielle 1 et lance la 2
```

```
032       RESET INERT_REC
033       GET MEM_3
034       GIVE INERT MEM_4
035       SET INERT_REC

          // début de la deuxième acquisition
036       GIVE REC MEM_2
037       SET IMG_REC
038       SET START_ACQ
039       WAIT ACQUIRING 1
040       RESET START_ACQ

          // attente de la fin de la deuxième acquisition
041       WAIT ACQUIRING 0
042       RESET IMG_REC
043       GET MEM_2

          // attente de la fin de l'envoi des données image 1 vers le host
044       WAIT BURSTING 0
045       GET MEM_1

          // chargement des registres et envoi des données inertielles 1 vers le host
046       GIVE BURST MEM_3
047       LOAD AD_BASE_BURST #0
048       LOAD NUMBER_BURST #600
049       SET START_BURST
050       WAIT BURSTING 1
051       RESET START_BURST

          // attente de la fin de l'envoi des données inertielles 1 vers le host
052       WAIT BURSTING 0
053       GET MEM_3

          // début de la deuxième partie de la boucle

          // début de la première intégration
054       LOAD INT_TIME #200000        //temps d'intégration de 10ms
055       SET START_INTEG
056       WAIT INTEGRATING 1
057       RESET START_INTEG

          // chargement des registres et envoi des données image 2 vers le host
058       GIVE BURST MEM_2
059       LOAD AD_BASE_BURST #0
060       LOAD NUMBER_BURST #1000000
061       SET START_BURST
062       WAIT BURSTING 1
063       RESET START_BURST

          // attente de la fin de la première intégration
064       WAIT INTEGRATING 0

          // arrête l'acquisition inertielle 2 et lance la 1
065       RESET INERT_REC
066       GET MEM_4
067       GIVE INERT MEM_3
068       SET INERT_REC

          // début de la première acquisition
069       GIVE REC MEM_1
070       SET IMG_REC
071       SET START_ACQ
072       WAIT ACQUIRING 1
073       RESET START_ACQ
```

```
        // attente de la fin de la première acquisition
074     WAIT ACQUIRING 0
075     RESET IMG_REC
076     GET MEM_1

        // attente de la fin de l'envoi des données image 2 vers le host
077     WAIT BURSTING 0
078     GET MEM_2

        // chargement des registres et envoi des données inertielles 2 vers le host
079     GIVE BURST MEM_4
080     LOAD AD_BASE_BURST #0
081     LOAD NUMBER_BURST #600
082     SET START_BURST
083     WAIT BURSTING 1
084     RESET START_BURST

        // attente de la fin de l'envoi des données inertielles 2 vers le host
085     WAIT BURSTING 0
086     GET MEM_4

        // retour de boucle
087     JUMP $21
        // fin du programme
```

Annexe D - Programme assembleur de l'application de détection de mouvements

```
        //début du programme

        //chargement du registre d'intégration
000     LOAD INT_TIME #200000       //temps d'intégration de 10ms

        //chargement des registres d'acquisition
001     LOAD ADDRESS_X #1344        //1024 + size_x/2
002     LOAD ADDRESS_Y #1264        //1024 + size_y/2
003     LOAD SIZE_X #640            //images de taille VGA
004     LOAD SIZE_Y #480
005     LOAD AD_BASE_REC #0         //stocker a partir de l'adresse 0

        //chargement des registres d'envoi
006     LOAD AD_BASE_BURST #0
007     LOAD NUMBER_BURST #307200

        //début de la première intégration
008     SET START_INTEG
009     WAIT INTEGRATING 1
010     RESET START_INTEG

        //attente de la fin de la première intégration
011     WAIT INTEGRATING 0

        //début de la première acquisition
012     GIVE REC MEM_1
013     SET IMG_REC
014     SET START_ACQ
015     WAIT ACQUIRING 1
016     RESET START_ACQ

        //début de la deuxième intégration
017     SET START_INTEG
018     WAIT INTEGRATING 1
019     RESET START_INTEG

        //attente de la fin de la première acquisition
020     WAIT ACQUIRING 0
021     RESET IMG_REC
022     GET MEM_1

        //attente de la fin de la deuxième intégration
023     WAIT INTEGRATING 0

        //début de la deuxième acquisition
024     GIVE REC MEM_2
025     SET IMG_REC
026     SET START_ACQ
027     WAIT ACQUIRING 1
028     RESET START_ACQ

        //début de la troisième intégration
029     SET START_INTEG
030     WAIT INTEGRATING 1
031     RESET START_INTEG

        //attente de la fin de la deuxième acquisition
```

```
032     WAIT ACQUIRING 0
033     RESET IMG_REC
034     GET MEM_2

        //début du traitement des données 1 & 2 -> 4
035     GIVE OP1 MEM_1
036     GIVE OP2 MEM_2
037     GIVE OP3 MEM_4
038     SET OPER_1_READY
039     WAIT OPER_1_BUSY 1
040     RESET OPER_1_READY

        //attente de la fin de la troisième intégration
041     WAIT INTEGRATING 0

        //début de la troisième acquisition
042     GIVE REC MEM_3
043     SET IMG_REC
044     SET START_ACQ
045     WAIT ACQUIRING 1
046     RESET START_ACQ

        //début de la première intégration
047     SET START_INTEG
048     WAIT INTEGRATING 1
049     RESET START_INTEG

        //DEBUT DE LA BOUCLE

        //attente de la fin du traitement des données 1 & 2 -> 4
050     WAIT OPER_1_BUSY 0
051     GET MEM_1
052     GET MEM_2
053     GET MEM_4

        //envoi des données 1 & 2 -> 4 vers le host
054     GIVE BURST MEM_4
055     SET START_BURST
056     WAIT BURSTING 1
057     RESET START_BURST

        //attente de la fin de la troisième acquisition
058     WAIT ACQUIRING 0
059     RESET IMG_REC
060     GET MEM_3

        //début du traitement des données 2 & 3 -> 5
061     GIVE OP1 MEM_2
062     GIVE OP2 MEM_3
063     GIVE OP3 MEM_5
064     SET OPER_1_READY
065     WAIT OPER_1_BUSY 1
066     RESET OPER_1_READY

        //attente de la fin de la première intégration
067     WAIT INTEGRATING 0

        //début de la première acquisition
068     GIVE REC MEM_1
069     SET IMG_REC
070     SET START_ACQ
071     WAIT ACQUIRING 1
072     RESET START_ACQ

        //début de la deuxième intégration
```

```
073     SET START_INTEG
074     WAIT INTEGRATING 1
075     RESET START_INTEG

        //attente de la fin de l'envoi de données 1 & 2 -> 4 vers le host
076     WAIT BURSTING 0
077     GET MEM_4

        //attente de la fin du traitement des données 2 & 3 -> 5
078     WAIT OPER_1_BUSY 0
079     GET MEM_2
080     GET MEM_3
081     GET MEM_5

        //envoi des données 2 & 3 -> 5 vers le host
082     GIVE BURST MEM_5
083     SET START_BURST
084     WAIT BURSTING 1
085     RESET START_BURST

        //attente de la fin de la première acquisition
086     WAIT ACQUIRING 0
087     RESET IMG_REC
088     GET MEM_1

        //début du traitement des données 3 & 1 -> 4
089     GIVE OP1 MEM_3
090     GIVE OP2 MEM_1
091     GIVE OP3 MEM_4
092     SET OPER_1_READY
093     WAIT OPER_1_BUSY 1
094     RESET OPER_1_READY

        //attente de la fin de la deuxième intégration
095     WAIT INTEGRATING 0

        //début de la deuxième acquisition
096     GIVE REC MEM_2
097     SET IMG_REC
098     SET START_ACQ
099     WAIT ACQUIRING 1
100     RESET START_ACQ

        //début de la troisième intégration
101     SET START_INTEG
102     WAIT INTEGRATING 1
103     RESET START_INTEG

        //attente de la fin de l'envoi de données 2 & 3 -> 5 vers le host
104     WAIT BURSTING 0
105     GET MEM_5

        //attente de la fin du traitement des données 3 & 1 -> 4
106     WAIT OPER_1_BUSY 0
107     GET MEM_3
108     GET MEM_1
109     GET MEM_4

        //envoi des données 3 & 1 -> 4 vers le host
110     GIVE BURST MEM_4
111     SET START_BURST
112     WAIT BURSTING 1
113     RESET START_BURST

        //attente de la fin de la deuxième acquisition
```

```
114     WAIT ACQUIRING 0
115     RESET IMG_REC
116     GET MEM_2

        //début du traitement des données 1 & 2 -> 5
117     GIVE OP1 MEM_1
118     GIVE OP2 MEM_2
119     GIVE OP3 MEM_5
120     SET OPER_1_READY
121     WAIT OPER_1_BUSY 1
122     RESET OPER_1_READY

        //attente de la fin de la troisième intégration
123     WAIT INTEGRATING 0

        //début de la troisième acquisition
124     GIVE REC MEM_3
125     SET IMG_REC
126     SET START_ACQ
127     WAIT ACQUIRING 1
128     RESET START_ACQ

        //début de la première intégration
129     SET START_INTEG
130     WAIT INTEGRATING 1
131     RESET START_INTEG

        //attente de la fin de l'envoi de données 3 & 1 -> 4 vers le host
132     WAIT BURSTING 0
133     GET MEM_4

        //attente de la fin du traitement des données 1 & 2 -> 5
134     WAIT OPER_1_BUSY 0
135     GET MEM_1
136     GET MEM_2
137     GET MEM_5

        //envoi des données 1 & 2 -> 5 vers le host
138     GIVE BURST MEM_5
139     SET START_BURST
140     WAIT BURSTING 1
141     RESET START_BURST

        //attente de la fin de la troisième acquisition
142     WAIT ACQUIRING 0
143     RESET IMG_REC
144     GET MEM_3

        //début du traitement des données 2 & 3 -> 4
145     GIVE OP1 MEM_2
146     GIVE OP2 MEM_3
147     GIVE OP3 MEM_4
148     SET OPER_1_READY
149     WAIT OPER_1_BUSY 1
150     RESET OPER_1_READY

        //attente de la fin de la première intégration
151     WAIT INTEGRATING 0

        //début de la première acquisition
152     GIVE REC MEM_1
153     SET IMG_REC
154     SET START_ACQ
155     WAIT ACQUIRING 1
156     RESET START_ACQ
```

```
        //début de la deuxième intégration
157     SET START_INTEG
158     WAIT INTEGRATING 1
159     RESET START_INTEG

        //attente de la fin de l'envoi de données 1 & 2 -> 5 vers le host
160     WAIT BURSTING 0
161     GET MEM_5

        //attente de la fin du traitement des données 2 & 3 -> 4
162     WAIT OPER_1_BUSY 0
163     GET MEM_2
164     GET MEM_3
165     GET MEM_4

        //envoi des données 2 & 3 -> 4 vers le host
166     GIVE BURST MEM_4
167     SET START_BURST
168     WAIT BURSTING 1
169     RESET START_BURST

        //attente de la fin de la première acquisition
170     WAIT ACQUIRING 0
171     RESET IMG_REC
172     GET MEM_1

        //début du traitement des données 3 & 1 -> 5
173     GIVE OP1 MEM_3
174     GIVE OP2 MEM_1
175     GIVE OP3 MEM_5
176     SET OPER_1_READY
177     WAIT OPER_1_BUSY 1
178     RESET OPER_1_READY

        //attente de la fin de la deuxième intégration
179     WAIT INTEGRATING 0

        //début de la deuxième acquisition
180     GIVE REC MEM_2
181     SET IMG_REC
182     SET START_ACQ
183     WAIT ACQUIRING 1
184     RESET START_ACQ

        //début de la troisième intégration
185     SET START_INTEG
186     WAIT INTEGRATING 1
187     RESET START_INTEG

        //attente de la fin de l'envoi de données 2 & 3 -> 4 vers le host
188     WAIT BURSTING 0
189     GET MEM_4

        //attente de la fin du traitement des données 3 & 1 -> 5
190     WAIT OPER_1_BUSY 0
191     GET MEM_3
192     GET MEM_1
193     GET MEM_5

        //envoi des données 3 & 1 -> 5 vers le host
194     GIVE BURST MEM_5
195     SET START_BURST
196     WAIT BURSTING 1
197     RESET START_BURST
```

```
        //attente de la fin de la deuxième acquisition
198     WAIT ACQUIRING 0
199     RESET IMG_REC
200     GET MEM_2

        //début du traitement des données 1 & 2 -> 4
201     GIVE OP1 MEM_1
202     GIVE OP2 MEM_2
203     GIVE OP3 MEM_4
204     SET OPER_1_READY
205     WAIT OPER_1_BUSY 1
206     RESET OPER_1_READY

        //attente de la fin de la troisième intégration
207     WAIT INTEGRATING 0

        //début de la troisième acquisition
208     GIVE REC MEM_3
209     SET IMG_REC
210     SET START_ACQ
211     WAIT ACQUIRING 1
212     RESET START_ACQ

        //début de la première intégration
213     SET START_INTEG
214     WAIT INTEGRATING 1
215     RESET START_INTEG

        //attente de la fin de l'envoi de données 3 & 1 -> 5 vers le host
216     WAIT BURSTING 0
217     GET MEM_5

        // retour de boucle
218     JUMP $50
        //fin du programme
```

A methodology for implementing vision applications on an heterogeneous Smart Camera platform

PhD thesis by Fábio Dias Real de Oliveira

Defended on July 6, 2010 - Blaise Pascal University
Clermont-Ferrand - France

Abstract

Smart Cameras are embedded artificial vision systems. These systems differ from "common" cameras due to their ability to analyze images and extract pertinent information about the observed scene, and doing this autonomously by using embedded processing resources. The application field for such systems is wide (CCTV, industrial vision, autonomous vehicles, etc.), but their implementation is quite complex and requires a high expertise level and long development times.

The work presented in this thesis deals with this problem, and proposes a design methodology which helps to simplify the application implementation process into FPGA-based smart camera platforms. This methodology is based upon a custom soft-core processor (instantiated in the FPGA), and a design flow allowing to deal separately with hardware issues dependent on the platform, and software issues dependent on the application.

Keywords: computer vision, smart camera, FPGA, design methodology, SoPC, real-time system, embedded system, soft-core processor.

Conception d'une méthodologie d'implémentation d'applications de vision dans une plateforme hétérogène de type Smart Camera

Thèse de doctorat présentée par M. Fábio Dias Real de Oliveira

Soutenue le 6 juillet 2010 à l'Université Blaise Pascal
Clermont-Ferrand - France

Résumé

Les caméras intelligentes, ou Smart Cameras, sont des systèmes embarqués de vision artificielle. Ces systèmes se différencient des caméras "communes" par leur capacité à analyser les images, afin d'en extraire des informations pertinentes sur la scène observée, et ceci de façon autonome grâce à des dispositifs embarqués de calcul. Les applications pratiques de ce type de système sont nombreuses (vidéo-surveillance, vision industrielle, véhicules autonomes, etc.), mais leur implémentation est assez complexe, et demande un haut degré d'expertise et des temps de développement élevés.

Les travaux présentés dans cette thèse s'adressent à cette problématique, et proposent une méthodologie d'implémentation permettant de simplifier le développement d'applications au sein des plateformes Smart Camera basées sur un dispositif FPGA. Cette méthodologie s'appuie d'une part sur l'instanciation au sein du FPGA d'un processeur "soft-core" taillé sur mesure, et d'autre part sur un flot de design à deux niveaux, permettant ainsi de traiter séparément les aspects matériels liés à la plateforme et les aspects algorithmiques liés à l'application.

Mots-clefs : vision par ordinateur, smart camera, FPGA, méthodologie d'implémentation, SoPC, système temps-réel, système embarqué, processeur soft-core.